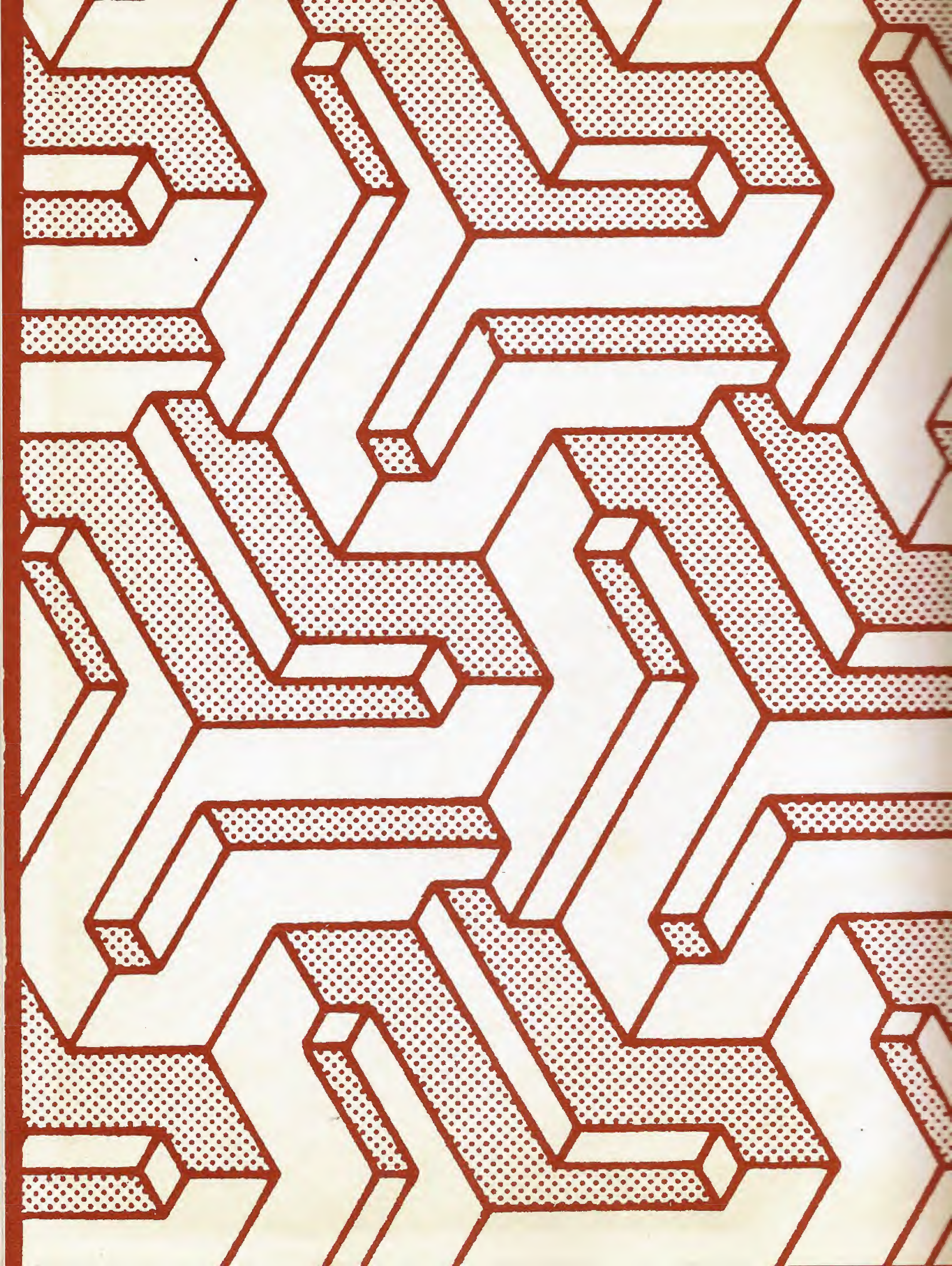
# GRAN ENCICLOPEDIA INFORMATICA

## LENGUAJES/2

EDICIONES NUEVA LENTE



# GRAN ENCICLOPEDIA INFORMATICA

EDICIONES NUEVA LENTE


GESTION
CONTROL
ALMACENES
FACTURACION
CONTABILIDAD
GENERAL

# SUMARIO

*Una publicación:*

**Ediciones Nueva Lente, S. A.**

*Director editor:* MIGUEL J. GOÑI

*Director de producción:* SANTOS ROBLES.

*Director de la obra:* FRANCISCO LARA.

*Colaboradores:* PL/3 - MANUEL MUÑOZ - ANGEL MARTINEZ - MIGUEL DE ROSENDO - DAVID SANTAOLALLA - SANTIAGO RUIZ - LUIS COCA - MIGUEL ANGEL VILA - MIGUEL ANGEL SANCHEZ VICENTE ROBLES.

*Diseño:* BRAVO/LOFISH.

*Maquetación:* JUAN JOSE DIAZ SANCHEZ.

*Ilustración:* JOSE OCHOA.

*Fotografía:* (Equipo Gálata) ALBINO LOPEZ y EDUARDO AGUDELO.

*Ediciones Nueva Lente, S. A.:*
Dirección y Administración:
Benito Castro, 12. 28028 Madrid. Tel.: 245 45 98.

*Números atrasados y suscripciones:*
Ediciones Ingelek, S. A.
Plaza de la Rep. Ecuador, 2 - 1.°. 28016 Madrid.
Tel.: 250 58 20.

*Plan general de la obra:*
18 tomos monográficos de aparición quincenal.

*Distribución en España:*
COEDIS, S. A. Valencia, 245. Tel.: 215 70 97.
08007 Barcelona.

*Delegación en Madrid:*
Serrano, 165. Tel.: 411 11 48.

*Distribución en Argentina:*
*Capital:* AYERBE
*Interior:* DGP

*Distribución en Chile:* Alfa Ltda.

*Distribución en México:*
INTERMEX, S. A.
Lucio Blanco, 435
México D.F.

*Distribución en Uruguay:*
Ledian, S. A.

*Edita para Chile:*
PYESA
Doctor Barros Borgoño, 123
Santiago de Chile

*Importador exclusivo Cono Sur:*
CADE, SRL. Pasaje Sud América, 1532.
Tel.: 21 24 64. Buenos Aires - 1.290. Argentina.



*Fotomecánica:* Ochoa, S. A.
Miguel Yuste, 32. 28037 Madrid.

*Impresión:* Gráficas Reunidas, S. A.
Avda. de Aragón, 56. 28027 Madrid.

ISBN de la obra: 84-7534-184-5.
ISBN del tomo 12: 84-7534-232-9

*Printed in Spain*
Depósito legal: M. 27.605-1986



Precio de venta al público en Canarias, Ceuta y Melilla: 940 ptas.
Marzo 1987

# LOGO (1)

## El lenguaje de la escuela



El lenguaje LOGO fue desarrollado por Seymour Papert en el Laboratorio de Inteligencia Artificial del M. I. T. (Massachusetts Institute of Technology). Papert y su grupo buscaban un sistema que resultara sencillo para la enseñanza del uso de los ordenadores. Tras varios meses de investigación, surgió el LOGO: un lenguaje moderno nacido bajo la inspiración del LISP. Este último es un lenguaje evolucionado, orientado a aplicaciones de inteligencia artificial, que en 1959 vio la luz desde los propios laboratorios del M. I. T.

Las bases conceptuales del LOGO son la sencillez, unida a la potencia de operación. Otras características fundamentales del LOGO se concretan en su naturaleza de lenguaje modular e interactivo. La modularidad se refleja en el hecho de que el usuario del LOGO va creando pequeños módulos independientes que, a su vez, servirán de piezas para construir estructuras más complejas. El LOGO es un lenguaje interactivo en el sentido de que cualquier orden es procesada de inmediato, con sólo introducirla en el ordenador.

Debido a sus orígenes, inspirados en un lenguaje para el tratamiento de listas de datos —el LISP (LISt Processing)—, uno de los puntos fuertes del LOGO reside en la manipulación de listas alfanuméricas. No obstante, su característica más espectacular hay que buscarla en el tratamiento de gráficos, tarea encomendada al sistema denominado TURTLE GRAPHICS.

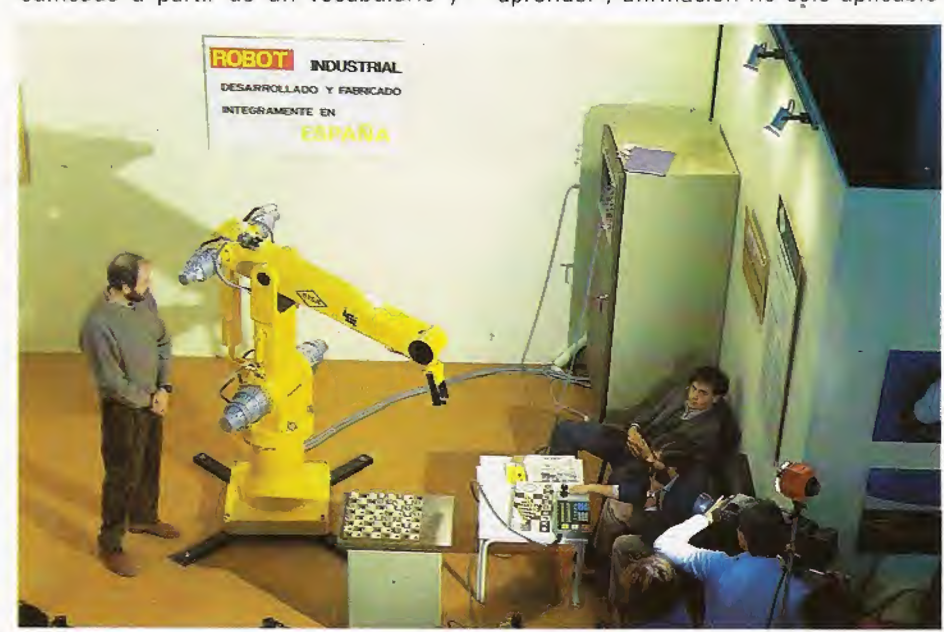
El popular TURTLE GRAPHICS (gráficos creados por medio de la tortuga), es un método fácil y divertido para la confección de dibujos. El concepto de *tortuga* procede de ciertos robots desarrollados a principios de 1960. Estas máquinas consistían en un caparazón flanqueado por ruedas (de ahí el nombre de tortuga) y funcionaban asociadas a un ordenador. El ordenador guiaba a la tortuga y ésta iba dejando marcada la huella de su trayectoria en el suelo.

A principios de 1970 surgió la variante actual de tan simpático y útil personaje: la tortuga que evoluciona sobre la pantalla. La idea de guiar a una tortuga a través de la pantalla de visualización, ha sido adoptada por otros muchos lenguajes para resolver el aspecto de creación de presentaciones gráficas. Junto a las posibilidades gráficas, el LOGO aporta un potente lenguaje de programación de tareas, fácil de aprender y edificado a partir de un vocabulario y unas estructuras simples y versátiles.

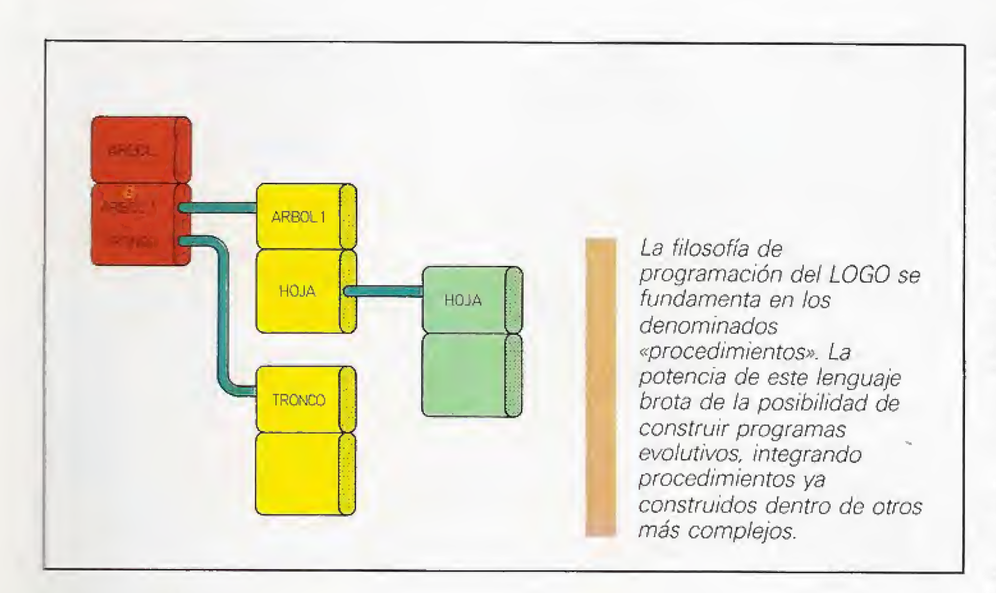
La sencillez inherente al empleo del LOGO, han convertido a este lenguaje en idóneo para labores pedagógicas y de iniciación a la programación de ordenadores. Los seguidores del LOGO declaran que se trata de «un lenguaje para aprender»; afirmación no sólo aplicable



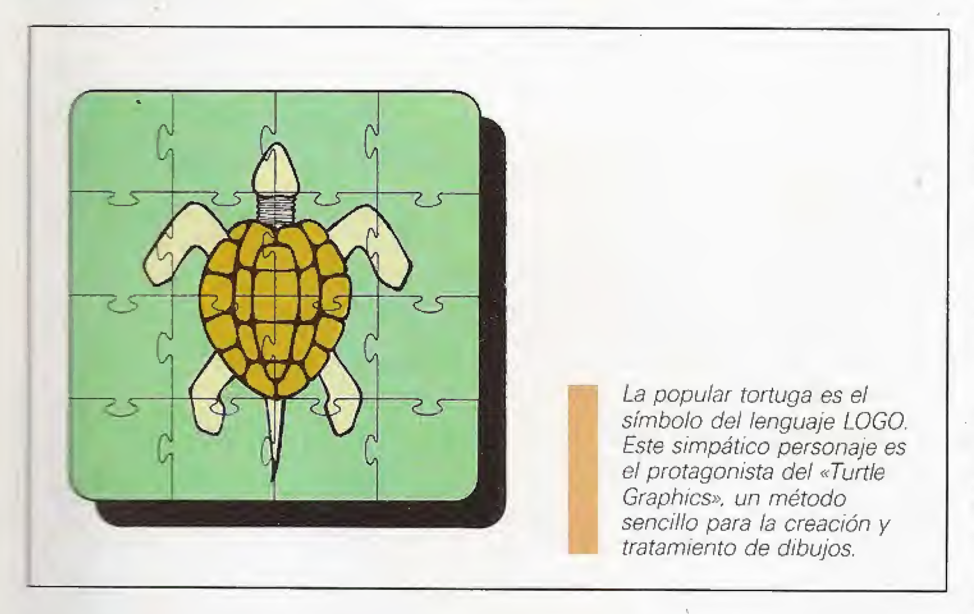

*Tras el BASIC, el LOGO es el lenguaje más popularizado en el terreno de los equipos domésticos. Son muchos los ordenadores personales de esta categoría que disponen de un intérprete del lenguaje LOGO.*




*El nacimiento del LOGO tuvo fuerte inspiración en el LISP, un lenguaje evolucionado orientado a aplicaciones de inteligencia artificial. Dado su origen y sus propias características, el LOGO es un lenguaje que resulta adecuado en el marco de la robótica.*




*La filosofía de programación del LOGO se fundamenta en los denominados «procedimientos». La potencia de este lenguaje brota de la posibilidad de construir programas evolutivos, integrando procedimientos ya construidos dentro de otros más complejos.*



*La popular tortuga es el símbolo del lenguaje LOGO. Este simpático personaje es el protagonista del «Turtle Graphics», un método sencillo para la creación y tratamiento de dibujos.*

a *aprender a programar,* sino también extensiva a *aprender a discurrir* en base a estructuras lógicas.

En efecto, el LOGO es algo más que un lenguaje de programación; hay que concebirlo como un verdadero *entorno para la enseñanza asistida por ordenador.* Con este lenguaje, los niños aprenden a trabajar con ordenadores como si se tratara de un simple juego. El secreto reside en la naturaleza interactiva del diálogo, que facilita la creación y depuración de estructuras por el sistema de prueba-error.

Existen en el mercado distintas versiones de LOGO. La mayor parte, desarrolladas por fabricantes de ordenadores en colaboración con el Grupo Logo del M. I. T. (TI. Logo, Apple Logo, Atari Logo, etc.). Se diferencian en muy poco, concretamente en abreviaturas de palabras LOGO, en la inclusión del TURTLE GRAPHICS y en el tratamiento de listas. También cabe mencionar la existencia de una versión en castellano del TI Logo (versión de Texas Instruments).

## La filosofía del LOGO

La programación en LOGO se fundamenta en el empleo de *procedimientos* definidos por el usuario. Estos procedimientos son conjuntos de órdenes a los que se accede como si se tratara, realmente, de una orden única. A su vez, las órdenes están constituidas por cadenas de datos, operadores y comandos. La potencia del LOGO brota de la posibilidad de utilizar procedimientos dentro de otros procedimientos. Semejante característica, permite programar potentes operaciones partiendo de unidades elementales. El método a seguir consiste, sencillamente, en construir los *procedimientos* complejos a partir de la asociación de otros procedimientos más simples definidos con anterioridad.

En realidad, el LOGO carece del concepto de programa como tal, lo que se suple con el denominado «WORK SPACE» o *espacio de trabajo.* Este contiene los procedimientos y variantes definidos por el usuario; y, por supuesto, en él existirá, generalmente, un *superprocedimiento,* o procedimiento principal, que hace uso de otros subprocedimientos.

En un mismo espacio de trabajo pueden coexistir varios superprocedimientos independientes. Esto equivale a tener varios «programas» independientes en la memoria central.

Como ya se ha indicado, el LOGO es un lenguaje fundamentalmente interactivo. Los mensajes de error son claros y concisos. Por ejemplo, si a través del teclado se introduce la palabra «HOLA», seguida de una acción sobre la tecla RETURN (la orden para que el ordenador asimile el mensaje introducido), la máquina responderá con un significativo:

**I DON'T KNOW HOW TO HOLA**
*(¡Ignoro cómo realizar la acción HOLA!)*

Con este mensaje, el LOGO comunica al usuario que el procedimiento HOLA no está definido en el *espacio de trabajo.* Si el error deriva de la omisión de un dato de entrada dentro de una orden, el LOGO responderá esta vez con:

**NOT ENOUGHT INPUTS TO XXX**
*(¡No hay suficentes datos de entrada para cursar la orden XXX!)*

Otra capacidad notable del LOGO se manifiesta en el tratamiento de cadenas alfanuméricas. Sus potentes instrucciones permiten añadir, extraer y comparar elementos inmersos en las mismas sin ninguna dificultad.

Sin lugar a dudas, la característica más espectacular del LOGO se encuentra en el TURTLE GRAPHICS. Un método de dibujo que consiste en desplazar una tortuga a través de la pantalla; ésta irá creando el dibujo al dejar visible el rastro de su trayectoria en los sucesivos movimientos. Crear un dibujo se convierte, pues, en algo tan simple y divertido como guiar el desplazamiento de tan simpático personaje.

## Texto y dibujos en la pantalla

Para desarrollar el conjunto de posibles actividades que brinda el LOGO, éste admite tres tipos o formatos de pantalla:

- Pantalla de *texto*
- Pantalla de *gráficos* y
- Pantalla *partida*.

La propia denominación que recibe cada formato de pantalla revela su utilidad: la primera está destinada a la presentación de texto, la segunda a la visualización de gráficos y la tercera, denominada pantalla partida o fraccionada, comparte ambas posibilidades.

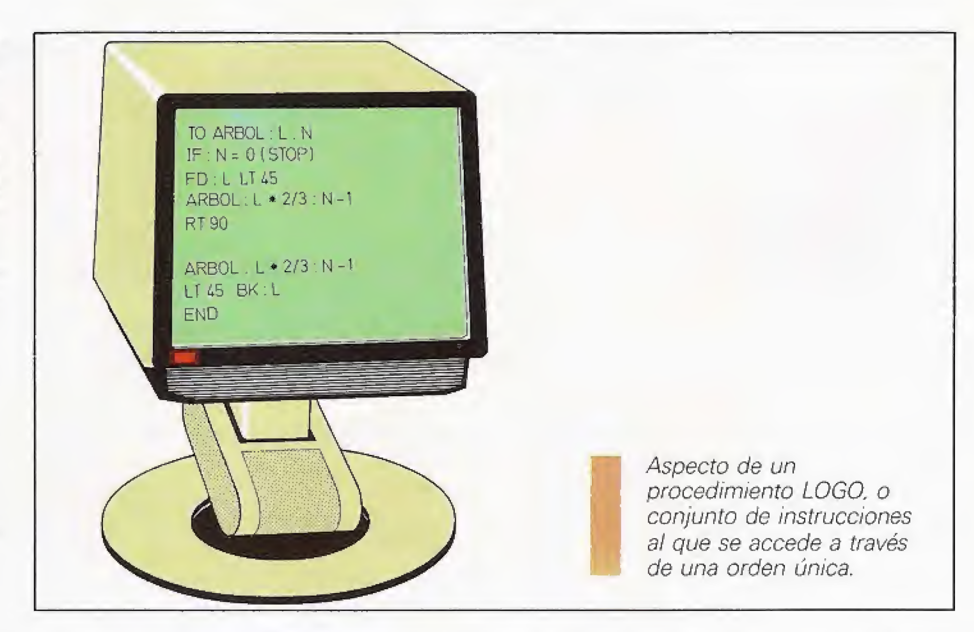
En la primera modalidad (pantalla de texto) no es posible ver a la tortuga y tampoco a los dibujos trazados con su colaboración. Este modo de presentación sólo resulta adecuado para la escritura de texto y suele ser el inicial, presente en el instante de conectar el ordenador.

Para acceder a la tortuga es preciso cambiar a otra modalidad de pantalla; para ello, basta con teclear un determinado comando gráfico. El efecto de tal acción será el salto a la pantalla de *gráficos* o a la pantalla *partida*.

Los comandos gráficos que permiten el «salto» a una nueva modalidad de pantalla son: FS, TS y SS.

El primero de ellos (FS: Full Screen) reserva toda la pantalla para gráficos. Si se introduce algún texto después de utilizar el comando FS, el usuario observará que su mensaje no aparece reflejado en ningún punto de la pantalla.

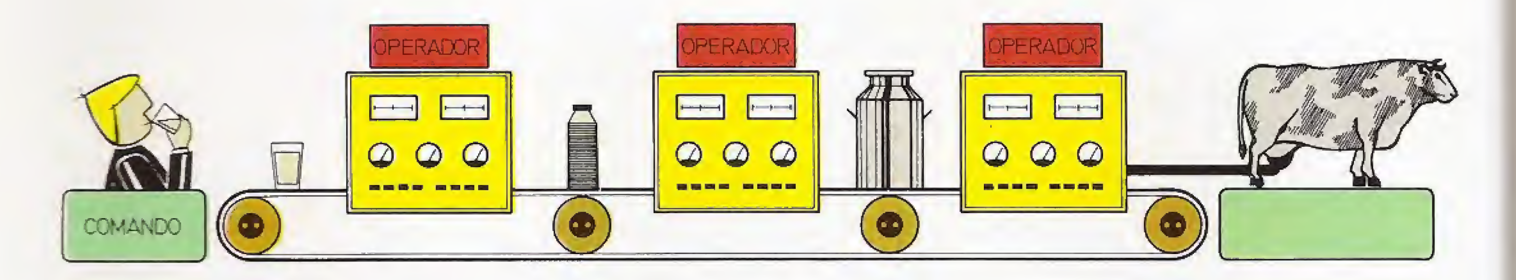
Si el comando introducido en SS (Split Screen), quedará seleccionada la pantalla partida: el texto aparecerá en la parte inferior de la misma, mientras que la




*Aspecto de un procedimiento LOGO, o conjunto de instrucciones al que se accede a través de una orden única.*




*El LOGO es uno de los lenguajes más frecuentes en ordenadores de tipo doméstico. Son muchos los modelos para los que existe el adecuado traductor de LOGO, como alternativa al intérprete BASIC habitual.*




*Estructura de una instrucción LOGO. El comando que define la acción a realizar se nutre de los datos de salida generados por los operadores que lo acompañan.*

zona superior quedará reservada a las evoluciones de la tortuga.

El tercero de los comandos, TS (Text Screen) destina toda la pantalla a la visualización de texto, sin dejar resquicio alguno para la tortuga.

Pasando alternativamente de Fs a TS, pueden ir observándose las órdenes introducidas y su ejecución en modo gráfico. Sin embargo, resulta más cómodo el uso de la pantalla partida, tal como tendremos ocasión de estudiar con detalle en los apartados dedicados a TURTLE GRAPHICS.

## Comandos y operadores

En el LOGO es muy importante el concepto de entrada y salida de datos. Imaginemos dos máquinas tragaperras, una de ellas expendedora de chicles y la otra un video-juego de «marcianos». En la primera, al introducir la moneda (dato de entrada), sale de inmediato un paquete de chicles (dato de salida). Por contra, al introducir la moneda (dato de entrada) en la máquina de video-juegos, no cae una nave galáctica por la ranura inferior (no hay dato de salida). Realmente, estamos otorgando la cualidad de «tangibles» a los datos de nuestro ejemplo, con lo cual, parece que no hay dato de salida en el segundo de los casos. Pasemos ahora a la concreción del LOGO. Para empezar, hay que tener en cuenta que las órdenes o instrucciones pueden ser de dos tipos:

- *comandos* u
- *operadores*

(según la terminología propia de este lenguaje).

La diferencia entre ambos tipos de instrucciones radica en que un *comando* es una orden que puede o no tener datos de entrada, pero que en ningún caso proporciona dato de salida (el ejemplo de la máquina de video-juegos). Por el contrario, el *operador* es una orden que proporciona siempre algún dato de salida (la máquina expendedora de chicles).

Por ejemplo, la orden FORWARD 5 es un comando: mueve la tortuga cinco espacios hacia adelante (5 es el dato de entrada), pero no proporciona ningún dato de salida. En cambio SUM 2 3, suma los números 2 y 3 entregando el número 5 como resultado. Las cifras 2 y 3 son datos de entrada, mientras que 5 es el dato de salida. Por lo tanto, SUM será un operador.

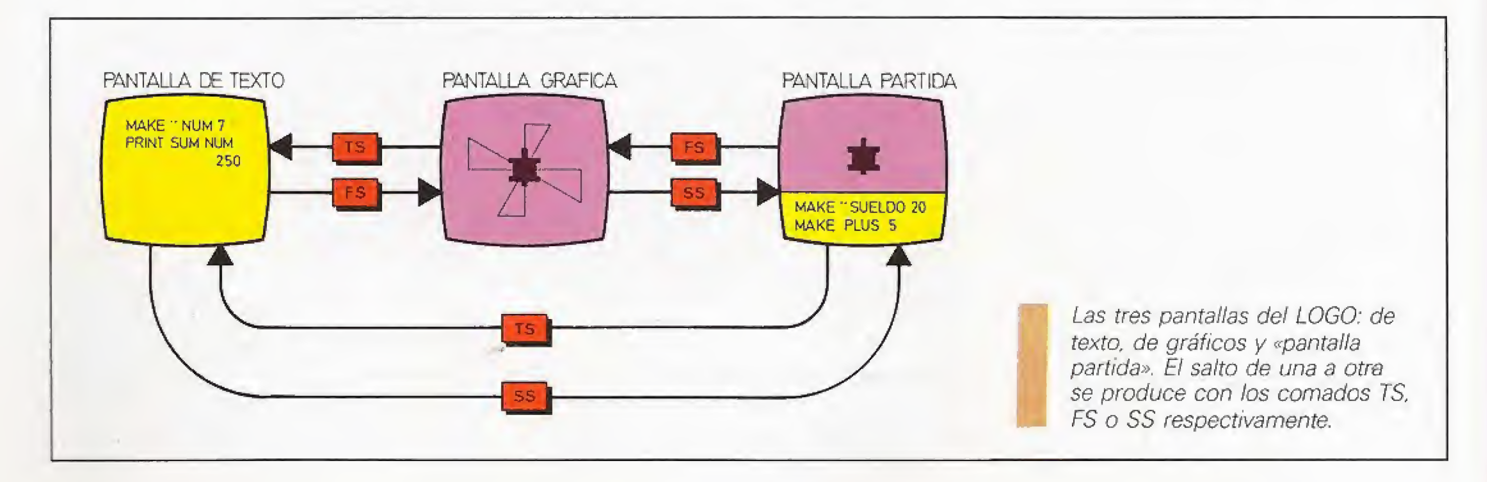
En el LOGO, las órdenes o instrucciones deben redactarse de tal forma que empiecen por un comando, puesto que los datos de entrada van siempre al final de la instrucción. Esta condición la observamos en cualquier orden, por ejemplo:

**FORWARD 5**

En la que el comando (FORWARD: hacia adelante) precede al dato de entrada (5). Los datos de entrada pueden coincidir, incluso, con los datos de salida de un operador. En la siguiente instrucción:

**FORWARD SUM 2 3**

Se ordena a la tortuga que avance un




*Las tres pantallas del LOGO: de texto, de gráficos y «pantalla partida». El salto de una a otra se produce con los comados TS, FS o SS respectivamente.*

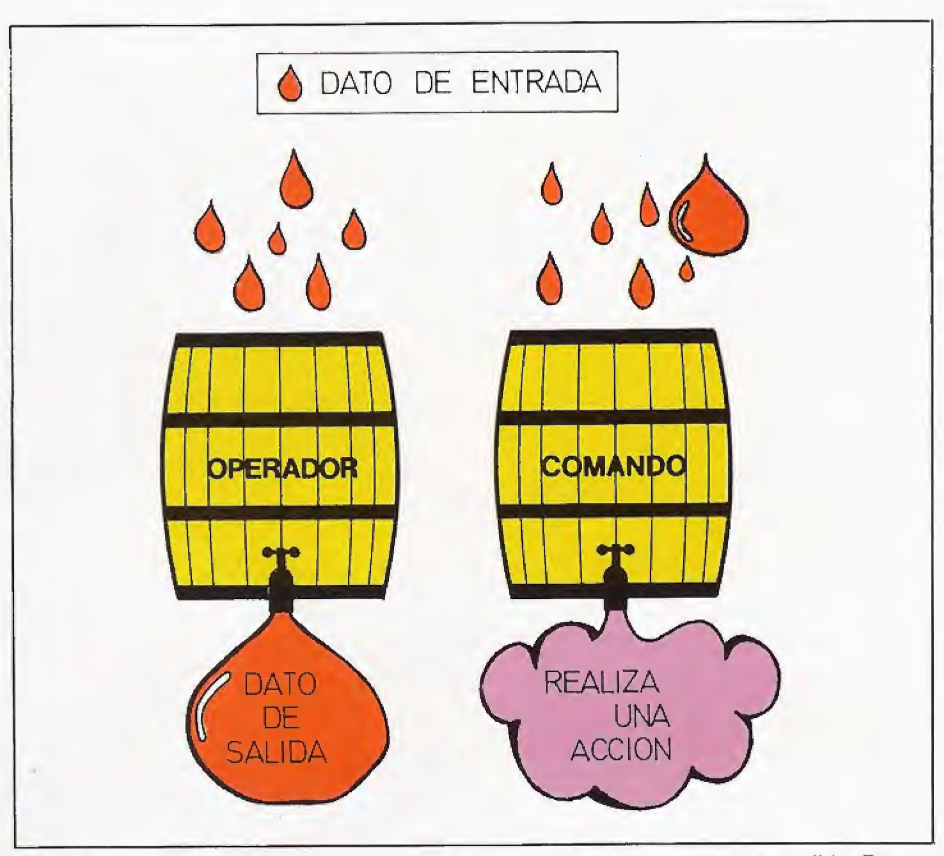
número de posiciones coincidente con el resultado de sumar 2 y 3. Por lo tanto, el dato de entrada del comando FORWARD es el dato de salida del operador SUM 2 3. A su vez, las entradas de SUM podrían coincidir con los datos de salida de otros operadores. En todo caso, hay que recordar que la primera palabra de la orden o instrucción debe ser un comando; de lo contrario, se perdería el dato de salida generado por el operador que lo sigue. Veamos un ejemplo. La instrucción siguiente:

```
SUM 284 2000
```

realiza la suma de los números indicados, sin embargo, el resultado no se utiliza para nada; no hay ningún comando al principio de la instrucción que le dé utilidad. En tal caso, el LOGO mostrará en la pantalla un mensaje de error:

```
SUM 284 2000
YOU DONT SAY WHAT
TO DO WITH 2284
```

(¡*No me has dicho qué debo hacer con el dato de salida 2284!).*




*Un operador LOGO recibe los datos de entrada y entrega un dato de salida. Este dato de salida puede pasar a formar parte de los datos de entrada de un comando destinado a realizar una acción específica.*

## Las variables del LOGO

Además de operadores, comandos y datos, existen otros elementos en las instrucciones LOGO: las *variables*.

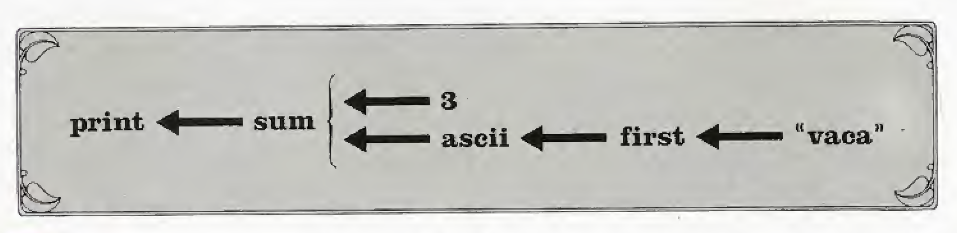
De forma simple, aunque ilustrativa, puede considerarse a la variable como un compartimento capaz de contener datos. En toda variable hay que distinguir dos partes:

- *nombre de la variable* y
- *contenido.*

El contenido es el valor del dato almacenado en dicho recipiente o *variable,* mientras que el nombre es la palabra que identifica a cada variable y la distingue por completo de las restantes.

Cabe suponer que las variables son algo semejante a los cajones de un armario clasificador. Cada cajón tiene una etiqueta que sirve para identificarlo y, por supuesto, dentro del cajón pueden almacenarse distintos objetos (datos). En el caso que nos ocupa, cada variable tiene un nombre que la identifica con exclusividad y en ella podemos almacenar muy diversos valores numéricos.

Por lo demás, podemos acceder al contenido de un cajón (al dato) a través del nombre que lo identifica (el nombre de la variable). Un buen ejemplo para ilustrar estos conceptos lo encontramos en el fichero de clientes de una empresa. El identificador de cada una de las fichas coincide con el nombre del cliente y, desde luego, es posible consultar una de estas fichas para obtener informa-




*Cualquier instrucción LOGO debe empezar por un comando. Si su lugar estuviera ocupado por un operador, la instrucción no ejecutaría una acción y se perderían los datos de salida de los operadores previos.*

ción. También se puede actualizar su contenido, ya sea borrando, añadiendo o sustituyendo datos. De forma análoga, también es posible consultar, modificar o borrar a voluntad el contenido o dato asignado a una variable.

Los datos de entrada incluidos en una orden puede definirse como el contenido de una variable, en lugar de expresarlos en forma de valor numérico fijo. Por ejemplo:

```
SUM :DATO 1 :DATO 2
```

suma el contenido de las variables DATO 1 y DATO 2.

Para definir cualquier variable, es preciso utilizar un comando al efecto: MAKE (hacer).

MAKE admite dos datos de entrada: el primero es el nombre con el que se desea identificar a la variable, mientras que el segundo coincide con el dato que se desea almacenar en la misma.

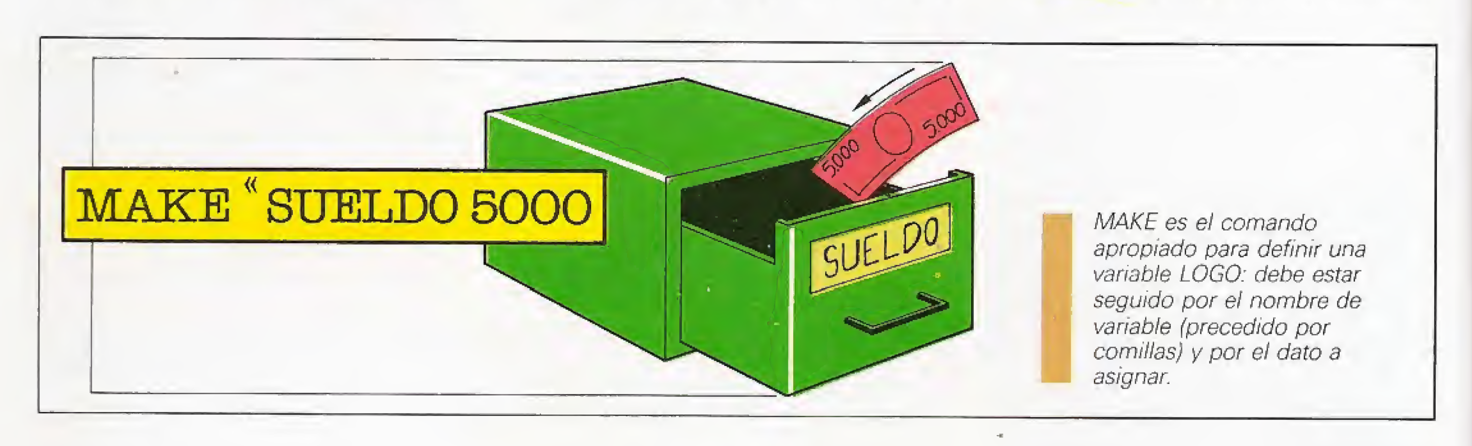
El nombre se puede elegir libremente, si bien, suele utilizarse una palabra relacionada con la naturaleza del contenido que va a almacenarse en dicha variable. Por ejemplo, si una variable va a contener un número correspondiente al sueldo que percibe un empleado, lo más lógico es otorgar a la variable en cuestión el nombre SUELDO. hay que precisar, que el nombre de la variable hay que introducirlo precedido por comillas ("). El motivo se comentará en el capítulo dedicado a las «palabras y listas» del LOGO.

El segundo dato que acompaña al comando MAKE debe coincidir con el valor que se desea otorgar a la variable. Más adelante, se verá que el dato en cuestión puede ser de varios tipos, incluyendo la propia salida de un operador o el contenido de otra variable. Por ejemplo:

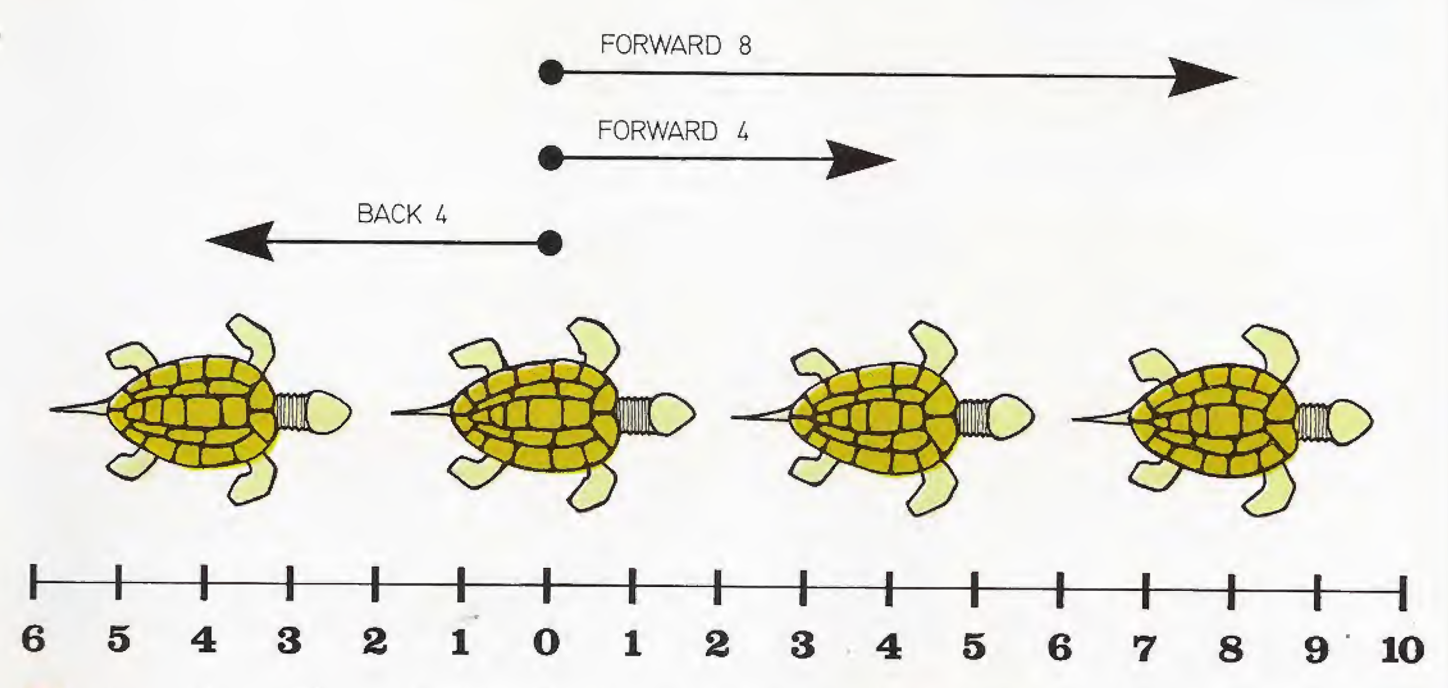
```
MAKE "DIA 30
```

asigna a la variable DIA el valor 30.

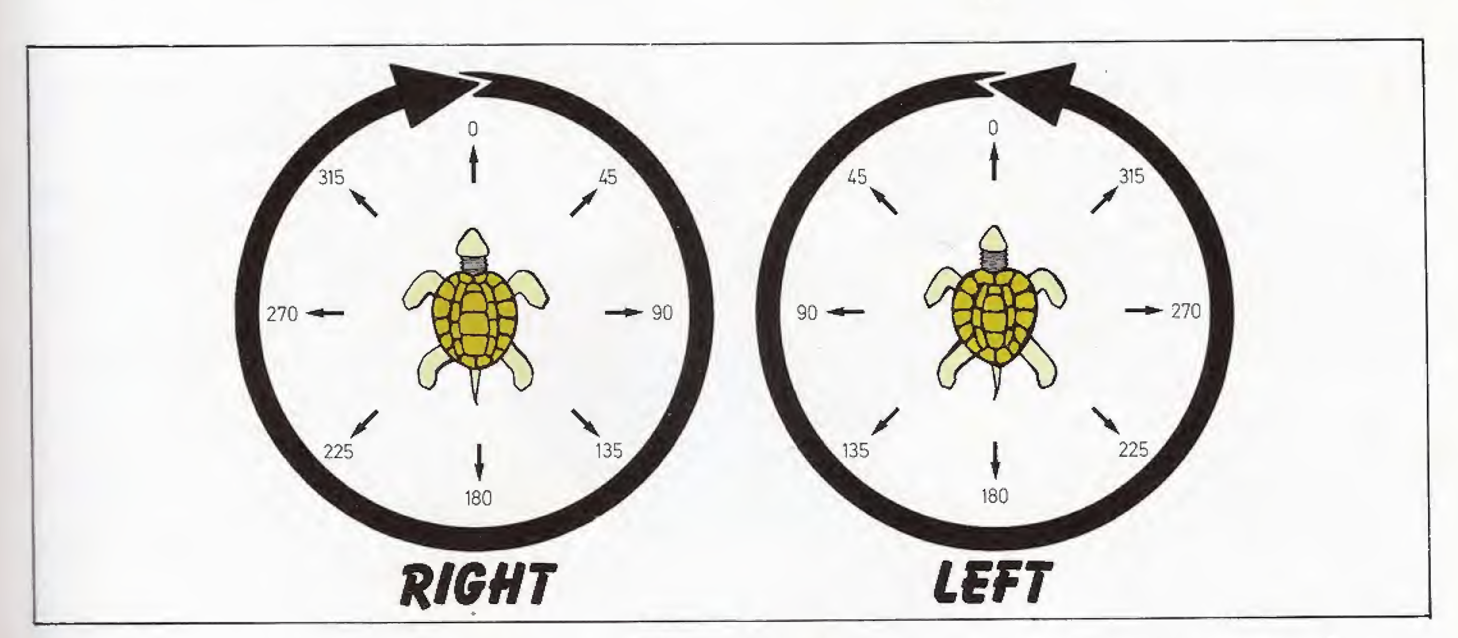
A la hora de acceder al contenido de




*MAKE es el comando apropiado para definir una variable LOGO: debe estar seguido por el nombre de variable (precedido por comillas) y por el dato a asignar.*




*Avance y retroceso (FORWARD y BACK). Estas dos simples acciones son el punto de partida de todo un vocabulario de órdenes que instruirán a la tortuga para la confección de dibujos en pantalla.*




*Dos son los comandos destinados a ordenar el giro de la tortuga: RIGHT (giro a la derecha) y LEFT (giro a la izquierda). En ambos casos, el comando correspondiente debe estar seguido por un número que exprese el ángulo de giro en grados.*

| ① | ② | ③ | ④ | ⑤ |
|---|---|---|---|---|
| POSICION INICIAL | RIGHT 45 | FORWARD 60 | BACK 120 | FORWARD 60 |

| ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|
| LEFT 270 | BACK 60 | FORWARD 120 | BACK 60 | LEFT 135 |

*Secuencia de órdenes que instruyen a la tortuga para que dibuje un aspa.*

la variable, bastará sencillamente con indicar el nombre de la misma, precedido por el signo «dos puntos» (:). Si la instrucción al efecto la empezamos con el comando PRINT:

```
PRINT :DIA
```

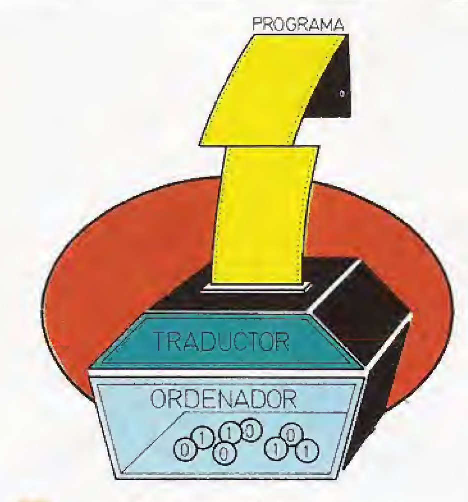
al ejecutar la orden, accionando la tecla RETURN, el contenido de la variable (DIA en nuestro caso) aparecerá escrito en la pantalla. El signo «dos puntos» que precede a la palabra DIA, indica al LOGO que se trata, precisamente, del nombre de una variable.

## Traductores de lenguajes

La comunicación entre dos personas que hablan distinto idioma no puede establecerse sin la colaboración de un traductor. Este puede ser un intérprete que realice una traducción simultánea, frase a frase; o simplemente un traductor que escriba en el idioma del destinatario el texto redactado en el idioma de origen. Esta es una realidad trasladable al mundo de los ordenadores.
Excepto en el caso de que el diálogo se mantenga utilizando el lenguaje máquina, es necesario un proceso de traducción para que el programa confeccionado por el usuario sea inteligible para el ordenador. Por supuesto, la tarea de traducción no correrá a cargo de traductores humanos; de ello se ocuparán los ordenadores una vez «instruidos» al efecto.
En cualquier proceso de traducción intervienen dos programas:

- *El programa fuente,* redactado en un lenguaje evolucionado, ensamblador o de alto nivel.
- *El programa objeto* o programa resultante del proceso de traducción, escrito en el lenguaje propio de la máquina.

Tal como ocurría en la traducción de comunicaciones humanas, también en este caso cabe una doble posibilidad: recurrir a un intérprete o traductor simultáneo, que permitirá un diálogo o comunicación interactiva, o utilizar los servicios de un traductor que reescriba el contenido de la comunicación en el lenguaje del destinatario (la máquina en nuestro caso). Los eficientes traductores humanos serán ahora




*La comunicación entre las personas que hablan distinto idioma no es posible sin la intervención de un traductor. Algo semejante ocurre a la hora de dialogar con las máquinas. Es preciso contar con un traductor que convierta al lenguaje íntimo de la máquina las órdenes y mensajes formulados por el usuario en un lenguaje de alto nivel.*

programas auxiliares especializados en tal menester. y, de nuevo, especializados en realizar una traducción interactiva o demorada (del mensaje en bloque).
Los *intépretes* son los programas especializados en la traducción interactiva. Traducen el programa línea a línea, de tal forma que el ordenador las ejecuta a medida que va disponiendo del resultado de la traducción.
La traducción diferida corre a cargo de los denominados programas *compiladores.* Estos traducen el programa fuente en bloque, obteniendo el correspondiente programa objeto redactado en el lenguaje de la máquina que debe ejecutarlo.
De la realidad humana podemos extraer otras conclusiones aplicables a los programas intérpretes y compiladores. En principio, la utilidad del intérprete se manifiesta en los diálogos interactivos; de ahí que este tipo de traductor resulte idóneo cuando se trata de habilitar una comunicación inmediata con la máquina. Otro factor significativo es que, dado el método de traducción, el intérprete invertirá bastante más tiempo en realizar su función que un compilador. Este último realiza la traducción del programa en bloque, de una sola vez, sin aguardar a que vayan ejecutándose las instrucciones a medida que son traducidas. La característica de velocidad se inclina, pues, hacia los compiladores. Este es un dato extensivo al proceso de ejecución: la ejecución del programa objeto, traducido en su integridad (compilado), es mucho más rápida que la ejecución línea a línea del programa interpretado (de tres a veinte veces más rápida).
La interpretación de un programa fuente se efectúa en el propio ordenador que cursará su ejecución. Sin embargo, el compilado de un programa puede no realizarse en el ordenador que debe ejecutar el programa objeto. Es frecuente que sea un ordenador

## Aprendiendo a andar

La herramienta que brinda el LOGO para crear dibujos es la *tortuga:* un simpático colaborador dispuesto a ejecutar los desplazamientos que le ordene el usuario. En sus evoluciones a través de la pantalla, ésta irá construyendo el dibujo al dejar una huella visible de su trayectoria. En efecto, es como si el inquieto personaje llevara una tiza adosada para perpetuar el rastro de sus movimientos. No cabe duda que la técnica resulta didáctica e incluso divertida, además de útil.

La mayor parte de las órdenes que entiende la tortuga son de tipo *comando.* Si bien, también existen operadores que reflejan las condiciones en las que está actuando la tortuga (posición, color, etc.). Más adelante se analizarán incluso algunas posibilidades avanzadas que amplían la eficacia de esta técnica: operación con varias tortugas simultáneamente, o transormación del aspecto de la tortuga a base de «disfrazarla».

Antes de empezar con el trazado de un dibujo, es ineludible observar el aspecto de la tortuga. Es importante distinguir la orientación de su cabeza. Este miembro es fundamental para el movimiento, puesto que indica la dirección en la que se desplazará el personaje. Una opción útil al respecto es pasar a la modalidad de pantalla partida, introduciendo a través del teclado la orden SS. La tortuga pasará a ocupar el centro de la pantalla. Situación en la que será inmediato comprobar cuál es su orientación.

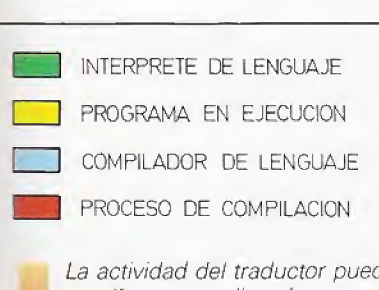
En el instante inicial, la tortuga debe encontrarse mirando hacia el límite superior de la pantalla. Si su disposición fuera otra, será preciso llevarla al estado inicial tecleando la orden CS.

Los desplazamientos más elementales de la tortuga son, naturalmente, los de avanzar o retroceder a lo largo de la dirección marcada por su eje longitudinal. Las órdenes que se ocuparán de instruirla al efecto son las siguientes:

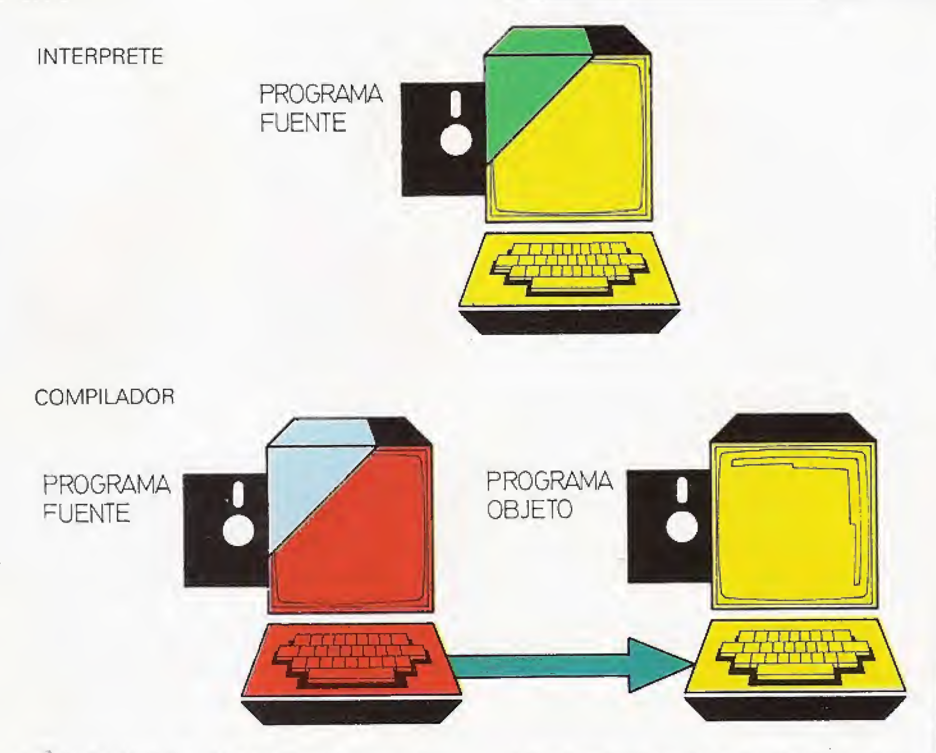
FORWARD: *avance*
BACK: *retroceso*

En ambos casos, ya sea el desplazamiento hacia adelante o hacia atrás, hay que indicar al quelonio el número de posiciones o «pasos» que debe dar antes de detenerse. Este número se introducirá tras el comando oportuno.

Por ejemplo, con la orden FORWARD 50, la tortuga se desplaza 50 posiciones hacia adelante, dejando impreso el rastro de su trayectoria. Si queremos que retroceda hasta ocupar de nuevo su posición original bastará con ordenar un desplazamiento hacia atrás del mismo número de posiciones: BACK 50. Si se introduce de nuevo la instrucción BACK 50, la tortuga retrocederá hacia el borde inferior de la pantalla visualizando su huella. Un nuevo FORWARD 50 devolverá a la tortuga al centro de la panta-




*La actividad del traductor puede manifestarse realizando una traducción simultánea del diálogo, o efectuando la traducción del mensaje total, de una sola vez. Los programas traductores del primer tipo son los denominados «intérpretes», y los del segundo «compiladores».*



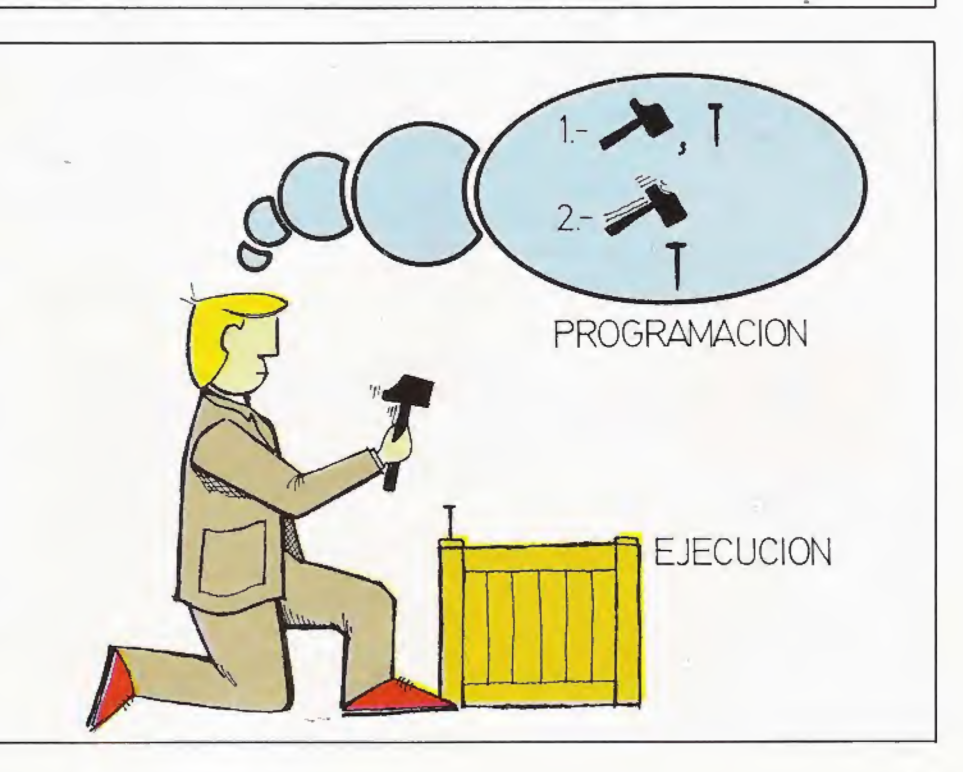

auxiliar el que se encargue de generar el programa objeto, redactándolo en el lenguaje máquina propio del equipo al que vaya destinado. Terminada la compilación, el programa puede ya introducirse y ejecutarse cuantas veces sea necesario en el ordenador de destino.
Son muchos los lenguajes para los que se dispone de ambos tipos de traductores.
La elección de intérprete o compilador dependerá del tipo de actividad que se encomiende al ordenador. Si la velocidad no es un factor primordial y prima la necesidad de mantener un diálogo interactivo, habrá que optar por un intérprete. Los microordenadores domésticos incorporan de origen un intérprete de lenguaje BASIC; si bien, el fabricante suele disponer de un catálogo de intérpretes y compiladores, en opción, de otros lenguajes.

Cuando la característica solicitada es una alta velocidad de ejecución, no cabe duda en la elección: hay que optar por un compilador.

Los lenguajes con una marcada inclinación hacia las aplicaciones interactivas (BASIC, LOGO, PILOT...) suelen utilizarse en versión compilada; mientras que los lenguajes más especializados, habitualmente no interactivos (FORTRAN, COBOL, ALGOL, PL/3...) actúan a través de compiladores.

## Programación y ejecución

Los lenguajes informáticos comparten el mismo objetivo que los humanos: ofrecer el medio adecuado para que pueda establecerse una comunicación. En el caso de la máquina, la comunicación se concreta en «instruirla» para que ésta desarrolle un trabajo o realice una tarea específica. Aquí es donde aparece el concepto de programa «o secuencia ordenada de instrucciones», cuya puesta en práctica o *ejecución* resuelve un cálculo, efectúa un tratamiento de información o, en general, realiza la tarea detallada en el programa.
En definitiva, *programar* al ordenador equivale a confeccionar la secuencia de instrucciones o programa, utilizando un lenguaje apropiado, inteligible para el ordenador.
Cuando la máquina deba efectuar el conjunto de operaciones o tarea programada, es preciso que el usuario introduzca el progama en la memoria del ordenador. A partir de ese instante, éste puede ordenar su *ejecución*. Ejecución que realizará el ordenador examinando las sucesivas instrucciones que componen el programa, interpretando su significado y cursando las órdenes y operaciones encomendadas.



lla, con lo que se habrá dibujado una línea vertical de 100 posiciones o unidades de desplazamiento: 50 hacia arriba y otras cincuenta por debajo de la posición original que ocupaba la tortuga. Pasemos a movimientos más complejos. ¿Es posible ordenar a la tortuga que cambie la dirección de desplazamiento? En efecto, basta sólo con instruirla para que realice el oportuno giro sobre sí misma. Los comandos al efecto son:

RIGHT: *derecha* y
LEFT: *izquierda*

Ambos comandos deben acompañarse de un dato de entrada que señale los grados del giro (recuerde que un giro de 360 grados equivale a una vuelta completa).

La orden RIGHT 90 dejará a la tortuga mirando hacia la derecha, mientras que un giro ordenado con LEFT 90, devolverá a la tortuga a la orientación de partida (apuntando hacia la zona superior de la pantalla). Supongamos que a partir de la posición de partida, ordenamos a la tortuga que ejecute la orden RIGHT 90 y tras ésta, la orden FORWARD 50. El resultado será la aparición en la pantalla de una línea horizontal de 50 puntos hacia la derecha, a partir del centro de la pantalla. Si queremos prolongar la línea horizontal por la izquierda, en otras cincuenta posiciones, será suficiente con introducir ahora la orden BACK 100.

¿Cómo devolver ahora a la tortuga a su posición de partida? Nada más fácil; aunque es necesario comunicarle dos órdenes: FORWARD 50 (regresa al punto central) y LEFT 90 (gira noventa grados para apuntar de nuevo al borde superior de la pantalla).

```
CS
FORWARD 50
BACK 100
HOME
RIGHT 90
FORWARD 50
BACK 100
HOME
LEFT 45
FORWARD 60
BACK 120
HOME
RIGHT 90
FORWARD 60
BACK 120
HOME
```

*Un nuevo ejemplo lo aporta el programa adjunto, cuya ejecución hará que la tortuga dibuje un asterisco sobre la pantalla.*

Sin lugar a dudas, el método es muy simple. La figura adjunta muestra la serie de órdenes que educarán a la tortuga para que dibuje un aspa; un ejemplo que resume el empleo de los cuatro comandos presentados.

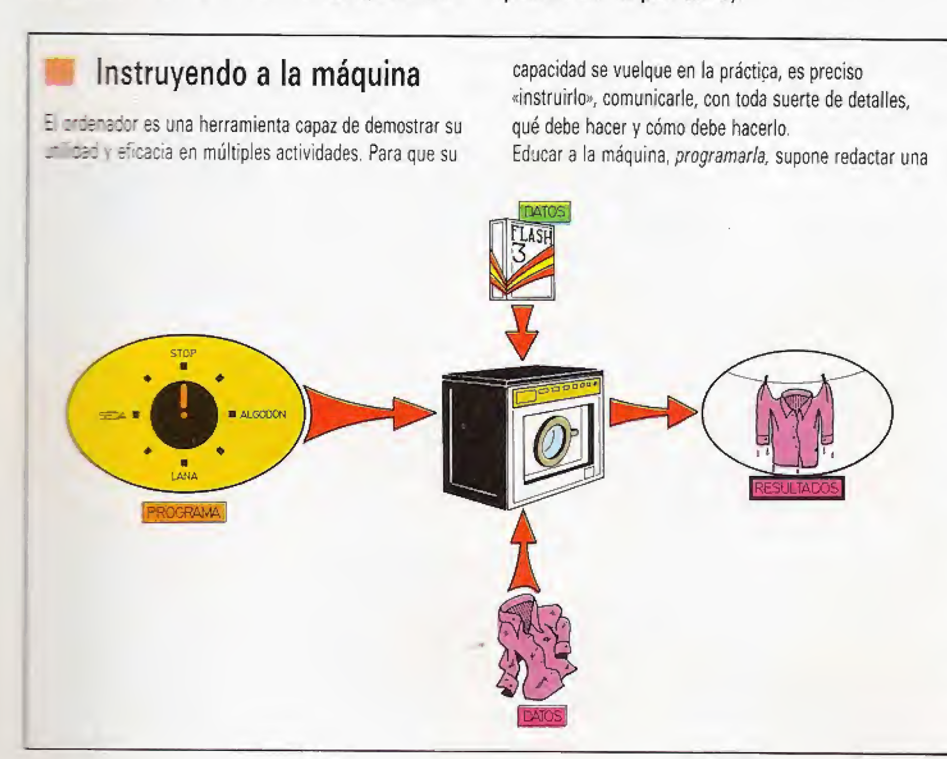
## Retorno al origen

Para devolver a la tortuga a la posición inicial, se han utilizado hasta aho-

### Instruyendo a la máquina

El ordenador es una herramienta capaz de demostrar su utilidad y eficacia en múltiples actividades. Para que su capacidad se vuelque en la práctica, es preciso «instruirlo», comunicarle, con toda suerte de detalles, qué debe hacer y cómo debe hacerlo.

Educar a la máquina, *programarla*, supone redactar una completa «receta» de instrucciones; o lo que es lo mismo, confeccionar un programa utilizando el lenguaje propio del ordenador.



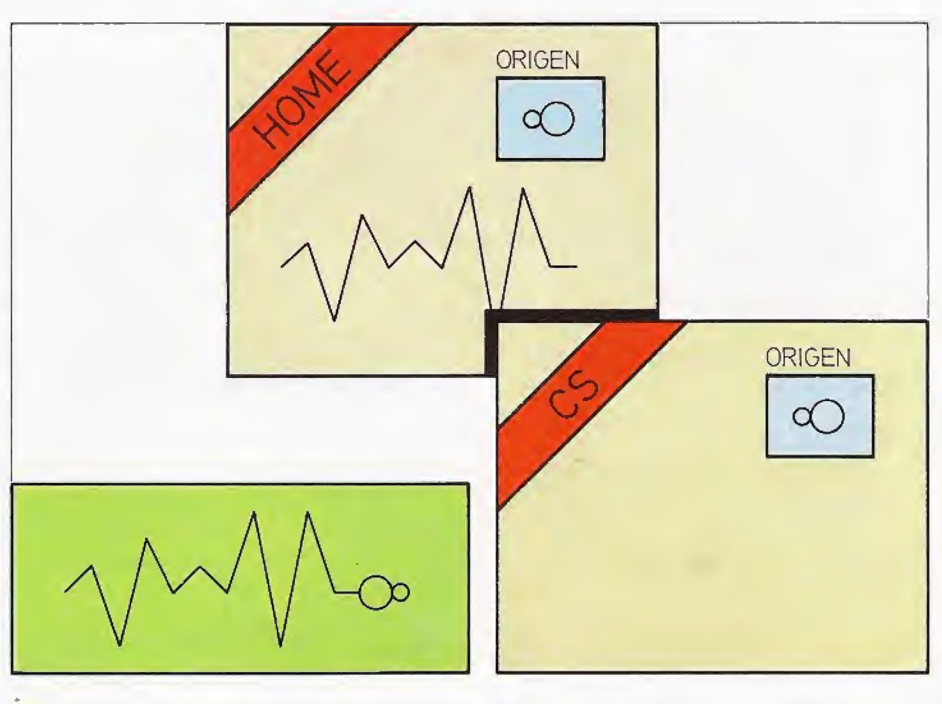

Si queremos que un cocinero nos prepare un plato que desconoce, es preciso darle la receta: una serie de instrucciones detalladas que, ejecutadas en su estricto orden, permitirán cocinar el plato. Hay ciertas instrucciones que deben ser especialmente detalladas; por el contrario, otras como «freír» o «pelar», son de sobra conocidas por el cocinero y no exigen mayores precisiones. Hay que tener cuidado para no cometer errores al escribir la receta, puesto que no es lo mismo «pollo frío» que «pollo frito».

Nuestro cocinero es el ordenador, alguien capaz de preparar cualquier plato que se nos antoje, por supuesto, siempre y cuando se le entregue la receta (el programa) adecuado.

Una vez que se conozca con detalle el lenguaje del ordenador, su aplicación a cualquier actividad práctica llegará a constituir una tarea tan cómoda y habitual como realizar la colada. Sabemos cómo aportarle los datos. El se ocupará de procesar los datos de acuerdo a las instrucciones del programa, y entregará el resultado con prontitud y precisión.

ra los comandos de movimiento y giro. Este es un método algo engorroso, puesto que las trayectorias de retorno dependen de la posición que ocupe la tortuga, distinta en cada caso. Un método bastante más cómodo para devolver al simpático quelonio a la posición de origen es la que brinda el comando HOME. Su ejecución obliga a la tortuga a regresar al origen, volviendo a la orientación de partida. Por supuesto, la línea que corresponde a la trayectoria de retorno quedará impresa en la pantalla.

Si lo que se desea es volver al principio borrando lo dibujado, hay que optar por otro comando CS (Clear Screen). Este elimina los rastros dejados por la tortuga y la coloca en su posición inicial. Uno de los gráficos adjuntos ilustra el funcionamiento de los nuevos comandos, en esta ocasión dibujando un asterisco.





*Las órdenes HOME y CS devuelven a la tortuga al origen de partida. La diferencia entre ambas radica en que CS borra la pantalla por completo, mientras que HOME no altera los dibujos que pudieran existir en la misma.*

## Pintando con la tortuga

Hasta ahora se han presentado algunos comandos que instruyen a la tortuga para que realice dibujos conexos; o lo que es lo mismo, dibujos continuos en los que todos los trazos están unidos entre sí. Sin embargo, también es posible mover a la tortuga sin dibujar. Para lograrlo, existen algunos comandos que permiten controlar la «tiza» con la que la tortuga pinta sobre la pantalla.

Para desplazar a la tortuga sin que ésta deje el rastro de su trayectoria, es necesario «levantar» la tiza. El comando PENUP es el que ordena tal acción. Una vez ejecutado, los siguientes movimientos de la tortuga no imprimirán trazo alguno en la pantalla.

Cuando haya que pintar de nuevo, habrá que ordenar a la tortuga que «baje la tiza». De ello se ocupa el comando PENDOWN. Este devuelve la tiza a su posición original, dispuesta para trazar la huella del desplazamiento sobre la pantalla. El siguiente ejemplo muestra el empleo de ambos comandos:

```
CS
PENUP
FORWARD 50
RIGHT 90
PENDOWN
FORWARD 50
PENUP
HOME
```

El programa ejemplo dibuja una línea en la zona superior de la pantalla, devolviendo a la tortuga a su posición inicial. Las sucesivas acciones de la tortuga al ejecutar el programa, empiezan tras borrar la pantalla (CS) y «levantar» la tiza (PENUP). Acto seguido (FORWARD 50), la tortuga avanzará 50 posiciones en sentido vertical sin imprimir huella. A continuación, girará 90° a la derecha (RIGHT 90) para, de inmediato, «bajar» la tiza (PENDOWN) y dibujar una línea horizontal de 50 posiciones al ejecutar la orden FORWARD 50. Para concluir su actividad, levantará de nuevo la tiza (PENUP) y regresará al original (HOME).

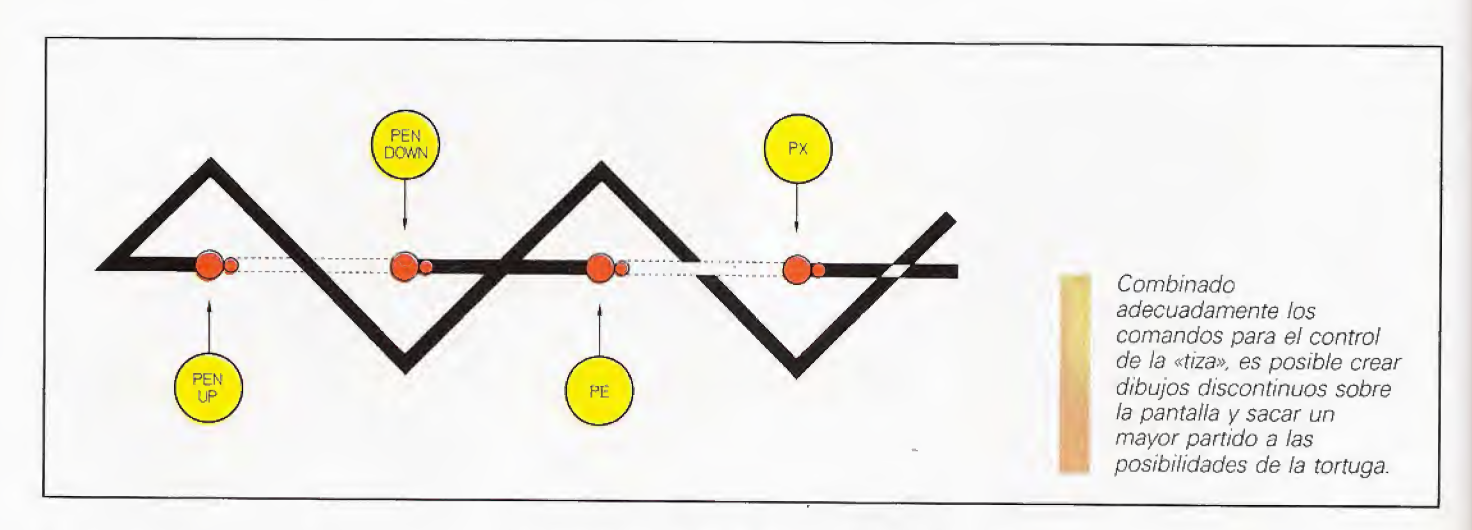
### Glosario

INTELIGENCIA ARTIFICIAL: rama de la ciencia que investiga la simulación de una auténtica inteligencia en un ordenador.
DEPURACION: fase del desarrollo de un programa consistente en la búsqueda de los fallos cometidos al codificarlo o escribirlo.
PROCEDIMIENTOS: subprogramas que forman parte de un programa mayor o «principal».
MEMORIA CENTRAL: zona para el almacenamiento de información residente en el interior del ordenador y a disposición directa de la unidad central de proceso. En ella se almacenan datos, variables y el programa o programas en curso.

Pero existen aún más posibilidades. El comando PE (Pen Erase) activa el borrador que la tortuga lleva consigo. Por donde pase, después de ejecutar la orden PE, irá borrando lo dibujado.

El último de los comandos de esta categoría es PX. Con este comando la tortuga borrará lo que encuentre pintado y pintará allí donde no encuentre trazos. Hay que tener en cuenta que el efecto de PX es el de alterar el color de tiza, de tal forma que siempre sea el opuesto al color de la zona sobre la que se desliza. En consecuencia, pintará trazos en «negativo», que al superponerse sobre las líneas dibujadas con la tiza en su color normal, las borrarán.

He aquí un nuevo ejemplo:




*Combinado adecuadamente los comandos para el control de la «tiza», es posible crear dibujos discontinuos sobre la pantalla y sacar un mayor partido a las posibilidades de la tortuga.*

```
PX
FORWARD 50
RIGHT 90
FORWARD 50
HOME
```

Al ejecutarlo, la tortuga recorrerá el mismo camino que en el ejemplo anterior; si bien, al estar activada ahora la función PX, trazará el «negativo» del dibujo previo y, en consecuencia, borrará la línea horizontal dibujada en el caso anterior.

El siguiente ejemplo recurre a PE para borrar parte del dibujo creado en el ejemplo ilustrativo de PX:

```
PE
FORWARD 25
RIGHT 90
FORWARD 25
HOME
```

Concretamente, la zona borrada coincidirá con la primera mitad de la línea vertical dibujada en el ejemplo anterior.

| TABLA DE ORDENES-LOGO | | |
|---|---|---|
| **Instrucción** | **Cometido** | **Operador/Comando** |
| FS | Selección de pantalla gráfica | Comando |
| TS | Selección de pantalla de texto | Comando |
| SS | Selección de pantalla partida | Comando |
| MAKE"<palabra> <dato> | Definición de variable | Comando |
| PRINT <objeto>* | Muestra el <objeto> por pantalla | Comando |

* <objeto>: puede ser una palabra, una lista, un número o una variable.

| TABLA DE ORDENES DEL «TURTLE GRAPHICS» | | |
|---|---|---|
| **Instrucción** | **Cometido** | **Operador/Comando** |
| FORWARD <número> | Avance de la tortuga | Comando |
| BACK <número> | Movimiento hacia atrás | Comando |
| RIGHT <grados> | Giro a la derecha | Comando |
| LEFT <grados> | Giro a la izquierda | Comando |
| HOME | Regreso al centro de pantalla | Comando |
| CS | Borrado de pantalla y retorno al origen | Comando |
| PENUP | «Levanta» la tiza | Comando |
| PENDOWN | «Baja» la tiza | Comando |
| PE | Activación del borrador | Comando |
| PX | Activación de tiza de color inverso (negativo) | Comando |
| PEN | Indica el estado de la tiza o modo de dibujo | Operador |
| CLEAN | Borra la pantalla dejando a la tortuga en la posición que ocupa | Comando |

## ¿Cuál es el estado de la tiza?

En cualquier momento, el programador puede perder la pista y no saber en qué modalidad está utilizando la tiza. Para solventar esta duda, se dispone de la función PEN.

PEN es un *operador* cuya respuesta señala el modo en el que está seleccionada la tiza. Los identificadores con los que responde tal operador constan de dos letras cuyo significado es:

PD: Modo PENDOWN
PU: Modo PENUP
PE: Modo Borrador
PX: Modo Inverso o «negativo».

Como quiera que PEN es un operador, habrá que utilizarlo precedido por un comando; por ejemplo: PRINT PEN, que escribirá en pantalla el identificador del modo en curso.

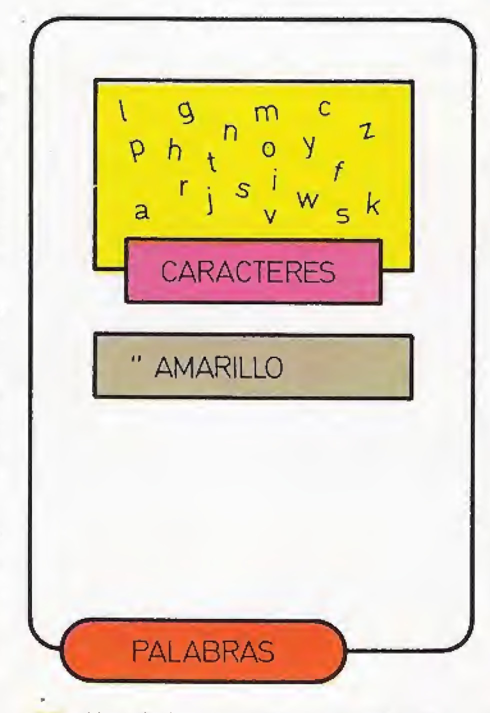
Por último cabe mencionar la existencia del comando CLEAN. Su función es limpiar la pantalla. Al igual que CS, borra todos los trazos anteriores. Pero a diferencia con aquél, no devuelve la tortuga a la posición de origen. En definitiva, CS equivale a la asociación de los comandos CLEAN y HOME.

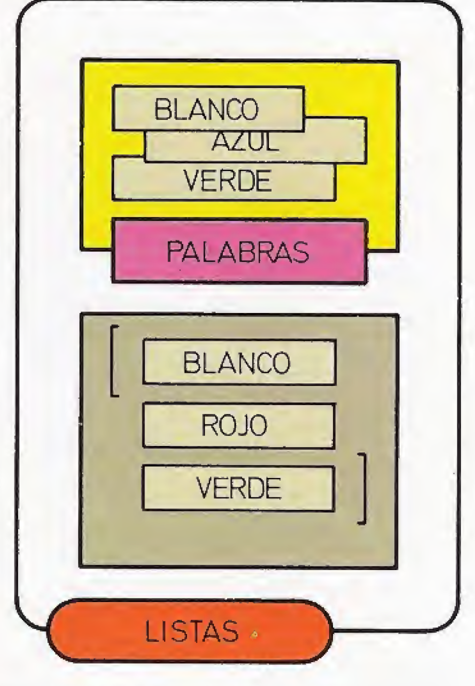
# LOGO (2)

## Los procedimientos



na de las facetas más importantes del LOGO es el tratamiento de palabras y listas. En la terminología de este lenguaje, una palabra es un conjunto de letras, números y/o signos especiales. Dada esta definición, es obvio que cualquier palabra en castellano (o en otro idioma) coincidirá con una palabra LOGO. Cabe recordar que ya en capítulos precedentes e han utilizado palabras para dar nombre a las variables. A su vez, una lista no es más que un conjunto de palabras. Una frase, un párrafo, o incluso un texto, son ejemplos de listas. El lenguaje LOGO permite manipular con notable soltura ambos tipos de elementos. En el tratamiento intervienen ciertos símbolos separadores que tienen asignada una específica función.



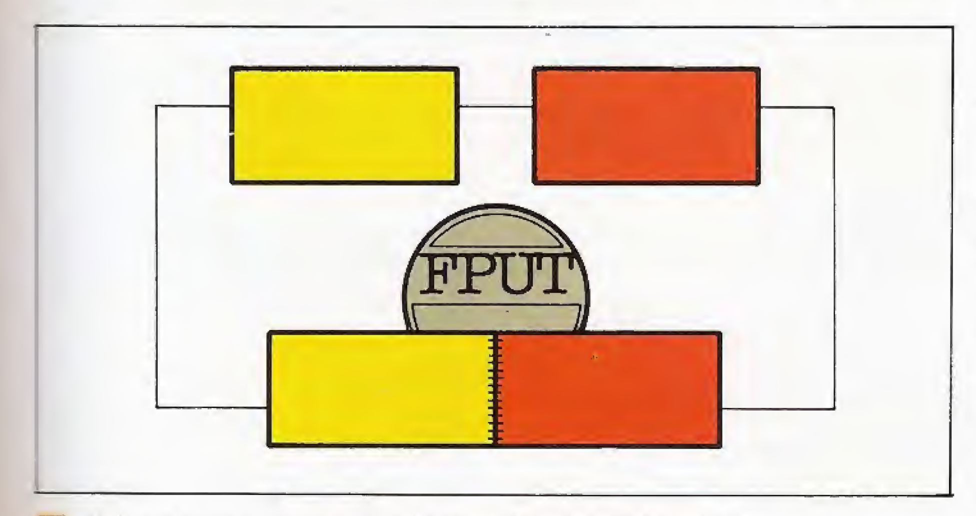

*Una de las aptitudes más relevantes del LOGO es el tratamiento de palabras y listas. Las palabras son agrupaciones de caracteres (letras, números y/o símbolos especiales autorizados), mientras que las listas están constituidas por conjuntos de palabras o de otras listas más elementales (sublistas).*

### Separadores

El separador es un símbolo especial cuya misión es delimitar las distintas partes de una «frase». El principal separador en LOGO es el espacio en blanco. Su presencia revela el fin de una palabra.

Si dos palabras se escriben una tras otra, sin interponer un espacio blanco entre ambas, el intérprete LOGO entenderá que se trata de una sola palabra. Así pues, UNOYDOS es una palabra única, UNO YDOS será interpretado como dos palabras, y UNO Y DOS como tres palabras independientes.

Un ordenador preparado para dialogar en LOGO no será capaz de asimilar, por ejemplo, la orden SUM32. Si en su lugar, se introduce la orden SUM 32, la máquina tomará como primera entrada 32 y será incapaz de encontrar el segundo dato de entrada. La instrucción correcta será, en definitiva, SUM 3 2. Este es un ejemplo que ilustra la importancia que tiene la correcta separación de las distintas zonas que intervienen en una instrucción.




*FPUT y LPUT son operadores cuya especialidad es añadir palabras o sublistas al principio o al final, respectivamente, de la lista que figure como segundo dato de entrada.*

### Comillas y corchetes

Una palabra puede ser un dato de entrada para una instrucción LOGO. En tal caso, la palabra ha de ir precedida por comillas. Para que el LOGO entienda que UNOYDOS es un dato de entrada, esta palabra debe aparecer como: "UNOYDOS. Por ejemplo, para escribir en la pantalla HOLA, la instrucción adecuada es:

PRINT "HOLA.

Cuando hay que tratar dos o más palabras juntas, éstas deben agruparse en la forma adecuada, formando una *lista:* conjunto de palabras encerradas entre corchetes. [BUENOS DIAS] es una lista




*El operador COUNT contabiliza el número de elementos que integran el dato de entrada sometido a examen. Este puede ser una palabra o una lista. En el primer caso se contarán los caracteres que conforman la palabra, mientras que en el segundo serán las palabras y sublistas los elementos contabilizados.*

y el ordenador la tratará como si se tratara de una unidad. En cambio, "BUENOS "DIAS son dos palabras independientes: no constituyen una lista y, en consecuencia, el ordenador las procesará por separado. por ejemplo:

PRINT [BUENOS DIAS]

escribirá en la pantalla la lista completa.

```
PRINT [BUENOS DIAS] (RT)
BUENOS DIAS
```

Una lista puede incluir, a su vez, otras listas de menor entidad o *sublistas*. Por ejemplo:

[HOLA [BUENOS DIAS]]

es una lista de dos elementos: una palabra y una sublista. La sublista en cuestión consta de dos elementos que en este caso coinciden con dos palabras simples. El concepto de elemento queda totalmente clarificado con el uso del *operador* COUNT (contar).

COUNT admite como dato de entrada una palabra o una lista, y devuelve como dato de salida el número de elementos que constituyen la entrada.

Así por ejemplo:

PRINT COUNT "HOLA (RT)
4

En efecto, el dato de salida o resultado es el número 4, puesto que son cuatro los elementos (caracteres) de la palabra examinada. Si en lugar de aplicar el operador COUNT a una palabra lo hacemos sobre una lista, los elementos a contar no serán ya caracteres, sino palabras y sublistas:

PRINT COUNT [BUENOS DIAS] (RT)
2

El resultado revela, en efecto, el número de palabras que intervienen en la lista señalada.

Tanto para [ ] (lista vacía) como para " (palabra vacía) el resultado de COUNT es 0.

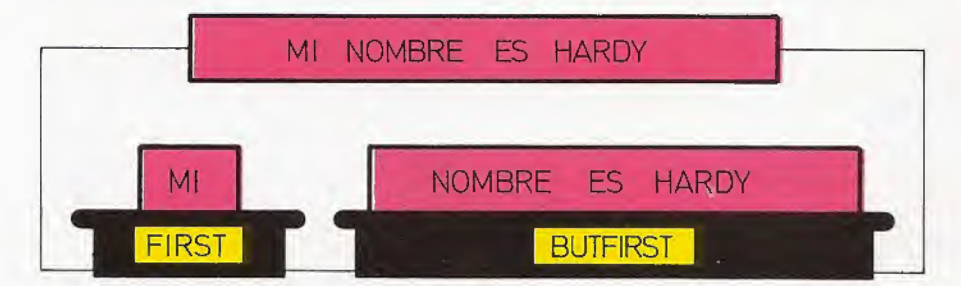
## Tratamiento de palabras y listas

El LOGO posee un nutrido grupo de operadores especializados en el tratamiento de palabras y listas. En primer lugar cabe mencionar los cuatro siguientes: FIRST, LAST, BUTFIRST y BUTLAST. Todos ellos son aplicables tanto a palabras como a listas.

FIRST y LAST devuelven, respectivamente, el primer y el último elemento del dato de entrada, mientras que los otros dos operadores hacen exactamente lo contrario; esto es: BUTFIRST elimina el primer elemento y BUTLAST el último. Veamos el siguiente ejemplo:

```
MAKE "LISTA1 [MINOMBRE ES]
PRINT FIRST :LISTA1
MI
PRINT LAST :LISTA1
ES
```

El programa empieza haciendo uso del comando MAKE para definir la LISTA1. Esta última palabra es el nombre que va a identificar a la variable, mientras que el dato a asignar (la lista de palabras) coincide con MI NOMBRE ES. Acto seguido aparece la instrucción PRINT FIRST :LISTA1, cuya finalidad es mostrar en la pantalla el resultado de




*FIRST es un operador LOGO que trunca su dato de entrada palabra o lista, tomando de éste su primer elemento. BUTFIRST realiza el mismo fraccionamiento, si bien entrega como salida todo los elementos del dato de entrada exceptuando el primero.*

aplicar el operador FIRST al contenido de la variable LISTA1 (cabe recordar la importancia de los dos puntos como indicador de variable). Por supuesto, el dato de salida es MI: la primera palabra o elemento de la lista.

La siguiente instrucción ordena la presentación en pantalla del último elemento de la referida lista, para lo cual se utiliza el operador LAST.

Los otros dos operadores presentados aplicados a la misma lista definida, darán como resultado el que muestra la siguiente pantalla:

```
PRINT BUTFIRST :LISTA1
NOMBRE ES

PRINT BUTLAST :LISTA1
MI NOMBRE
```

BUTFIRST presenta el contenido de la lista especificada, una vez suprimido el primer elemento, mientras que BUTLAST muestra la lista en cuestión exceptuando el último de sus elementos.

Las posibilidades de los cuatro operadores descritos no se reducen a la simple actuación independiente. Pueden asociarse para dar un tratamiento más complejo a los datos de entrada.

```
PRINT FIRST FIRST BUTLAST :LISTA1
M
```

Este es un ejemplo algo más complicado que combina la presencia de dos operadores FIRST y uno de tipo BUTLAST. Cabe recordar que su aplicación es extensiva a listas y a palabras, actuando sobre los elementos correspondientes: palabras en el caso de que se traten listas y caracteres en el caso de tratar palabras.

La instrucción del ejemplo aplica el operador BUTLAST al dato de entrada constituido por la lista definida (MI NOMBRE ES); el dato de salida será, en consecuencia, la lista MI NOMBRE. A su vez, esta lista actúa como dato de entrada de un operador FIRST, con lo que el nuevo dato de salida será la palabra MI. Por último, ésta constituye el dato de entrada del primer operador FIRST. Ahora, ya no es una lista el dato afectado por FIRST sino una palabra, de ahí que el resultado coincida con el primer elemento o carácter de la misma: M.

Dos nuevos operadores LOGO destinados al tratamiento de listas (en esta ocasión no son adecuados para operar con palabras), son FPUT y LPUT. Su cometido, ilustrado por el siguiente ejemplo, es añadir un elemento al principio o al final de la lista indicada.

```
PRINT FPUT "JUAN :LISTA1
JUAN MI NOMBRE ES

PRINT LPUT [JUAN PEREZ] :LISTA1
MI NOMBRE ES JUAN PEREZ
```

En el primer caso se inserta la palabra JUAN delante de la LISTA1, por medio del operador FPUT. La segunda instrucción, que ilustra la actuación de LPUT, añade la lista JUAN PEREZ al final de la indicada por medio de la variable LIST1.

Así como FPUT y LPUT sólo admiten listas como segunda entrada, el operador WORD admite únicamente palabras. Su misión es transformar a un conjunto de palabras en una palabra única. Así, por ejemplo, la salida de WORD "POR "QUE es la palabra PORQUE.

Si el número de entradas es superior a dos, es necesario encerrar entre paréntesis al operador WORD y a los datos de entrada que lo acompañan:

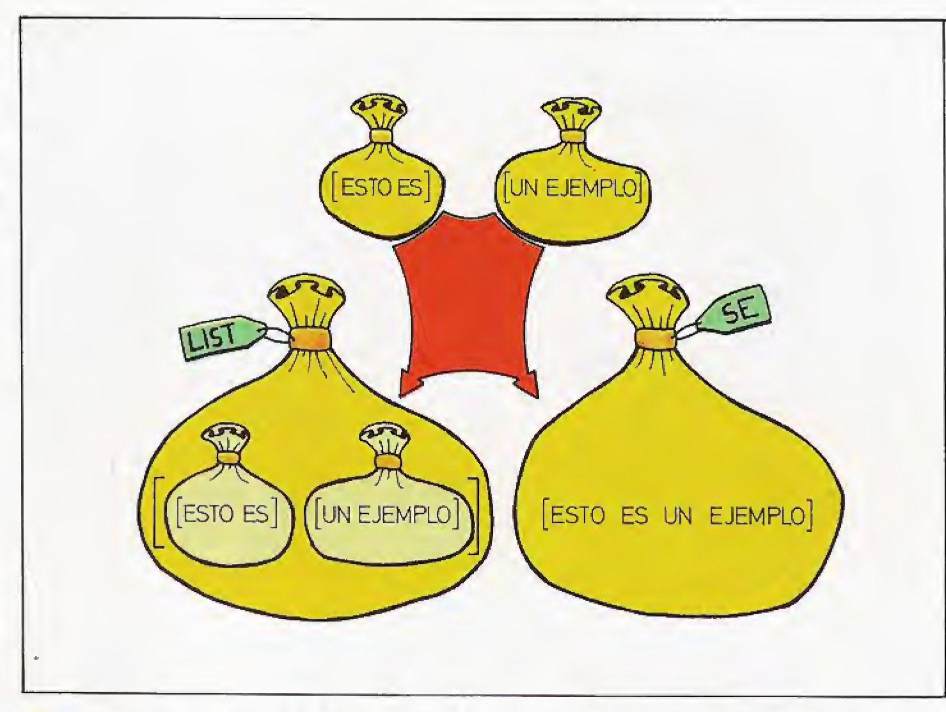
(WORD "TA "TE "TI "TO "TU)

Cuando el objetivo sea construir una lista, hay que optar por el empleo de LIST. Este *operador* construye una lista integrada por sus datos de entrada.

Al contrario de WORD, LIST sólo admite dos entradas; éstas pueden ser tanto palabras como listas.

```
MAKE "YO [JUAN PEREZ]
PRINT WORD "JUAN "PEREZ
JUANPEREZ

PRINT LIST :LISTA1 :YO
[MI NOMBRE ES] [JUAN PEREZ]
```




*La diferencia entre SE y LIST radica en que LIST asocia sublistas, mientras que SE construye listas finales agrupando las palabras de los datos de entrada.*



La modularidad del lenguaje LOGO queda plasmada en los procedimientos. Estos son módulos que pueden integrarse sucesivamente en otros procedimientos, cada vez más complejos, hasta construir el programa.

El ejemplo añade a la definición realizada anteriormente (LISTA1), la definición de una nueva variable de tipo lista con la que operar (YO). La segunda instrucción PRINT incluye al operador LIST, cuyo efecto es construir una nueva lista a partir de las dos sublistas especificadas como dato de entrada.

Otro de los operadores de esta categoría es SE (SEntence). Admite cualquier número de entradas, expresándolas al igual que en el caso de WORD.

SE sintetiza una lista resultante a partir de sus entradas si éstas son palabras, o a partir de los elementos de los datos de entrada si éstos son listas. En definitiva, la lista de salida de SE constará de palabras y nunca de sublistas como era el caso de LIST.

Una prueba palpable de tal distinción entre SE y LIST la aporta la inclusión del operador COUNT en las dos últimas instrucciones PRINT del siguiente ejemplo:

```
PRINT SE :LISTA1 :YO
MI NOMBRE ES JUAN PEREZ

PRINT COUNT LIST :LISTA1 :YO
2

PRINT COUNT SE :LISTA1 :YO
5
```

El primer COUNT aplicado a LIST cuenta las sublistas (2), mientras que el segundo cuenta las palabras resultantes (5).

Los párrafos precedentes resaltan la importancia que tiene distinguir perfectamente en las entradas y salidas cuándo se trata de palabras y cuándo de listas.

Si un operador recibe una entrada de tipo erróneo, el ordenador mostrará en la pantalla un mensaje de error. Por ejemplo, después de teclear: FPUT "IN"CREIBLE. Aparecerá el mensaje: FPUT DOESN 'T LIKE "CREIBLE AS INPUT (FPUT no admite "CREIBLE como entrada). Un ligero repaso al operador FPUT le hará reparar en que la segunda entrada no puede ser una palabra sino que debe ser una lista.

## El operador ASCII

El ordenador almacena y trata la información codificada a modo de palabras binarias o grupos de ceros y unos. Tanto las instrucciones como los datos (numéricos o alfanuméricos) adoptan este aspecto para que puedan ser manipulados por la máquina.

A la hora de proceder a la codificación, el ordenador puede acogerse a uno de los muchos códigos alfanuméricos, estandarizados o no, capaces de adecuar la información para que sea manipulable por los circuitos electrónicos. Entre ellos, el más extendido es el código ASCII (American Standard Code for Information Interchange). A pesar de su naturaleza de código estándar, existen distintas versiones del código ASCII, cuyas diferencias radican en la inclusión de un mayor o menor número de caracteres especiales; no obstante, las letras tienen siempre asignados los mismos códigos: del 65 (A) al 90 (Z). El LOGO dispone de operadores que permiten la conversión de carácter a código y viceversa. El primero de ellos, ASCII, devuelve el código del carácter que se especifique como dato de entrada.

La función opuesta corre a cargo del operador CHAR. Este transforma un número de entrada en el carácter ASCII correspondiente.

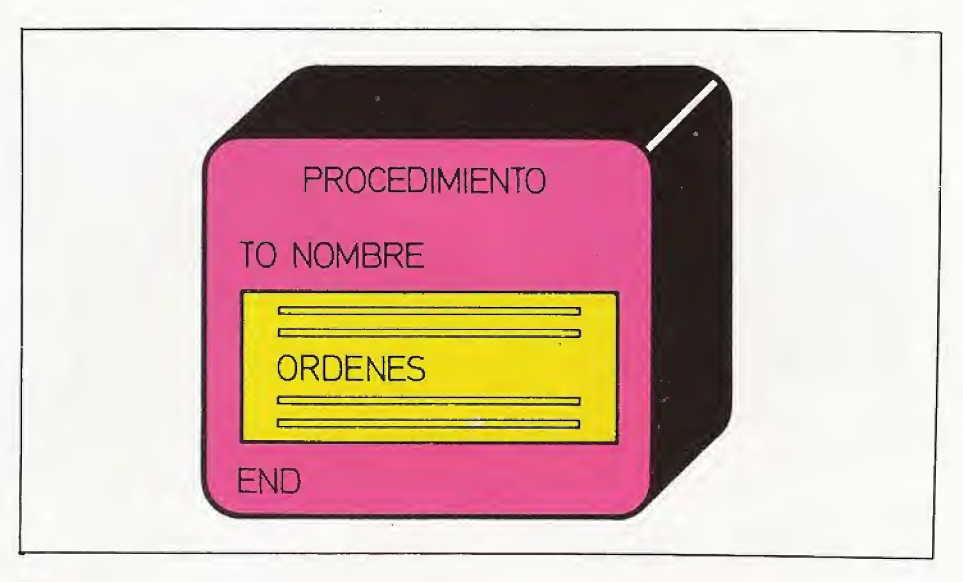
```
PRINT ASCII "A
65
PRINT CHAR 66
B
PRINT CHAR SUM 1 ASSCII "B
C
```

Los ejemplos anteriores ilustran la actuación de ambos operadores.

## Creación de procedimientos

A la hora de definir las características primordiales del LOGO, surge de inmediato el calificativo de modular. En efecto, los programas en LOGO están construidos a base de módulos cuya conjunción da lugar a estructuras más complejas. Estos módulos son los denominados procedimientos.

Un *procedimiento* no es más que una sucesión de órdenes encaminadas a realizar una acción más compleja. Requisito básico es que el conjunto de órdenes que constituyen el procedimiento aparezcan ordenadas en una secuencia apropiada; hay que tener en cuenta que no es lo mismo sumar dos cantidades y elevar el resultado al cubo, que elevar al cubo las cantidades y luego sumar los resultados. Las órdenes de un procedimiento, una vez ejecutadas, darán lugar a un resultado; éste coincidirá con la acción encomendada al procedimiento. Tal acción puede ser, sencillamente, una parte de un cálculo más complejo. Por ejemplo, un procedimiento que resuelva ecuaciones de segundo grado resultará de gran utilidad a la hora de hallar solución a múltiples problemas matemáticos.




*Los programas LOGO son estructuras creadas a partir de la asociación de procedimientos o módulos elementales. Un procedimiento es un conjunto de órdenes destinadas a programar una determinada acción.*




*Los procedimientos pueden asociarse para dar cuerpo a «superprocedimientos» o procedimientos LOGO más complejos.*

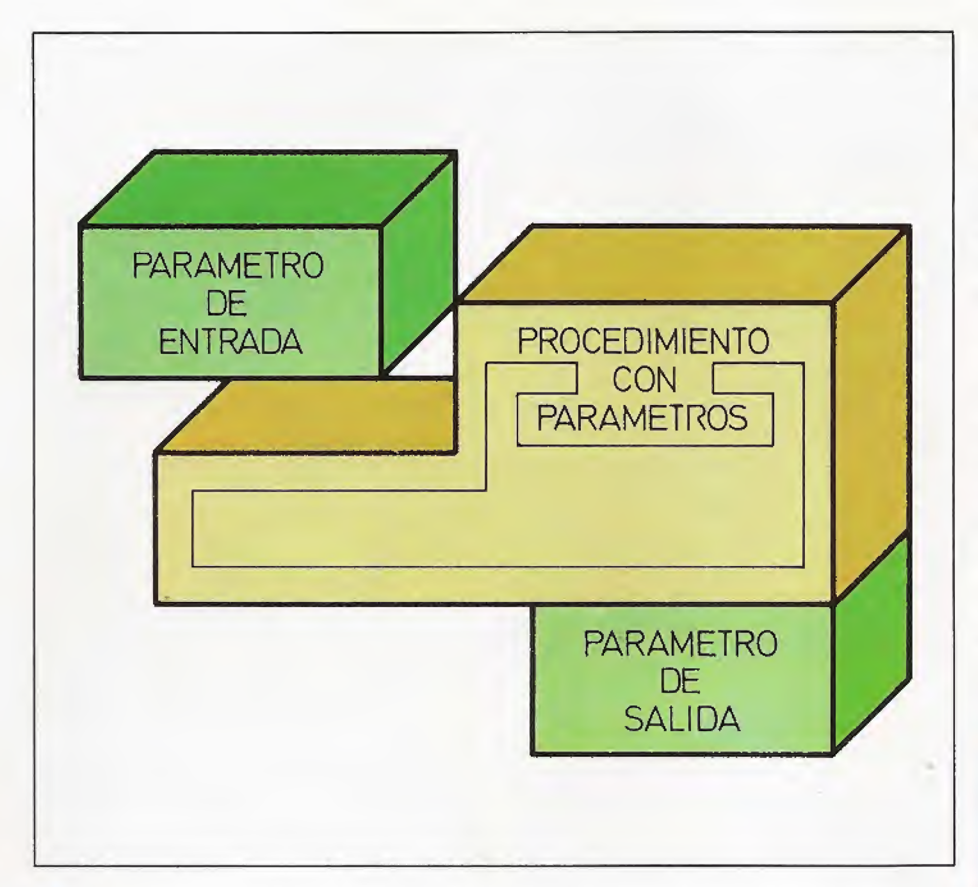
Todo procedimiento LOGO comienza con la palabra TO y finaliza con END. La primera línea debe contener, tras la palabra clave TO, el nombre con el que se va a identificar al procedimiento. Ello indica al LOGO que a partir de ese punto empieza el procedimiento especificado.

Las restantes líneas estarán ocupadas por las sucesivas órdenes, con una condición importante: la primera orden de cada línea debe ser un *comando.* En la última línea se colocará la palabra clave END, para que el LOGO entienda que ha concluido la definición del procedimiento.

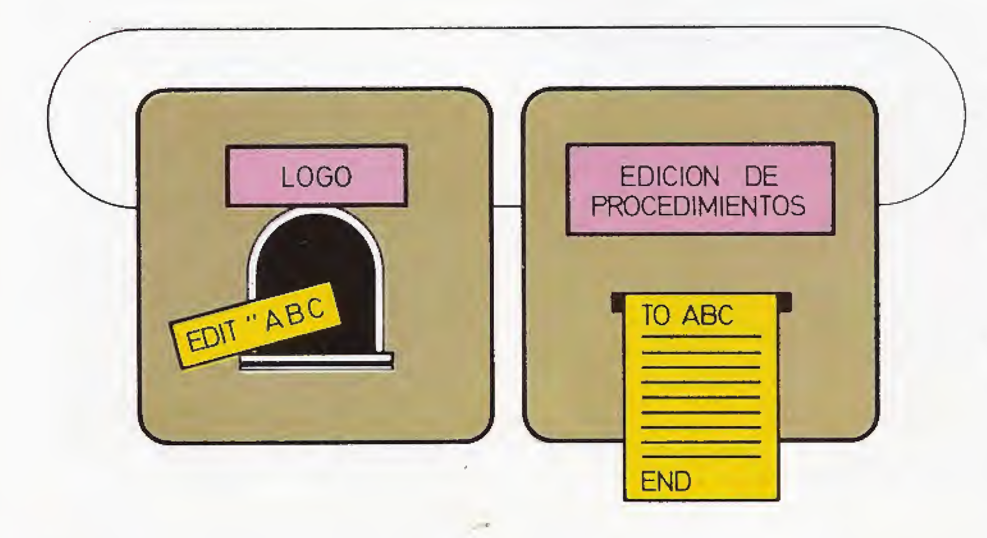
```
TO HOLA
PRINT "HOLA
PRINT [BIENVENIDO AL LOGO]
END

HOLA DEFINED
```

El programa corresponde a la creación de un procedimiento. En efecto, cabe observar que se han respetado las condiciones impuestas para la defini-




*Los procedimientos pueden exigir la presencia de parámetros de entrada que precisen su actuación al ejecutarlos. Asimismo, los procedimientos pueden emitir datos o parámetros de salida.*




*A la hora de modificar un procedimiento, no es preciso reescribirlo de nuevo en su totalidad. El lenguaje LOGO dispone de un comando de edición (EDIT) que visualiza el contenido del procedimiento elegido y permite su modificación.*

ción. La primera línea empieza con la palabra TO, seguida por el nombre procedimiento —HOLA, en nuestro caso—, y la última línea incluye la orden END; esta última comunica al ordenador que ha terminado la definición.

La respuesta de la máquina no se hace esperar: muestra en la pantalla el mensaje "HOLA DEFINED" (HOLA definido).

A partir de este preciso instante, para hacer uso del procedimiento HOLA, bastará con teclear su nombre.

Realmente, la creación de un procedimiento equivale a definir una nueva orden que funcionará exactamente igual que las incluidas en el repertorio original del LOGO. De hecho, un procedimiento puede incluso formar parte de otro procedimiento más complejo (superprocedimiento). Veamos un nuevo ejemplo:

```
TO SALUDO
HOLA
PRINT [COMO TE LLAMAS?]
PRINT LPUT FIRST RL [BUENOS DIAS]
END

SALUDO DEFINED
```

En esta ocasión se ha definido un nuevo procedimiento, denominado SALUDO. Un dato significativo es que se ha utilizado el procedimiento anterior, HOLA, como si se tratara de un *comando* propio del LOGO. Esta filosofía permite la subdivisión de problemas complicados en pequeños fragmentos elementales.

Por ejemplo, para calcular los gastos domésticos cabe seguir los siguientes pasos:

— Apuntar gastos de alimentación.
— Sumarlos y apuntar el resultado.
— Apuntar gastos de servicios contratados para la casa (luz, agua...).
— Sumar y escribir resultado.
— Apuntar gastos de otras compras.
— Sumar y apuntar resultado.
— Sumar los totales y escribir el resultado.




*La orden EDIT vuelca el procedimiento en la pantalla y brinda al usuario la posibilidad de corregir y alterar su contenido. La edición se reduce a desplazar el cursor al punto adecuado y utilizar las teclas alfabéticas y numéricas para realizar los cambios oportunos.*

La tarea aparece fraccionada en un conjunto de acciones elementales que, sin lugar a dudas, facilitan la comprensión y el cálculo. Esta misma filosofía de descomponer un trabajo en acciones parciales es compartida por la programación en LOGO. El caso propuesto puede adoptar la forma de procedimiento LOGO:

```
TO GASTOS
ENTRA.ALIMEN
SUMA.ALIMEN
ENTRA.CASA
SUMA.CASA
ENTRA.OTROS
SUMA.OTROS
SUMA.TOTAL
END
```

El procedimiento global GASTOS incluye las sucesivas acciones a realizar para evaluar el gasto doméstico total; desde luego, hace uso de otros procedimientos más elementales que será preciso definir por separado. En el ejemplo, se observa que todas las órdenes de GASTOS son procedimientos. Todas ellas deben ser creadas antes de utilizar el procedimiento GASTOS.

La definición del último de ellos, SUMA.TOTAL, puede adoptar la siguiente forma:

```
TO SUMA.TOTAL
MAKE "TOTAL SUM :OTROS :CASA
:ALIMENTOS
PRINT [TOTAL GASTOS]
PRINT LIST :TOTAL "PTS
END

SUMA.TOTAL DEFINED
```

Las variables OTROS, CASA y ALIMENTOS, utilizadas para calcular la suma total, contienen las sumas parciales de los respectivos conceptos. Por ejemplo, la variable :CASA que almacena el gasto total de los servicios domésticos contratados se calculará dentro del procedimiento SUMA.CASA:

```
TO SUMA.CASA
MAKE "CASA SUMLIST :LISTA.CASA
PRINT [TOTAL GASTOS DE CASA]
PRINT LIST :CASA "PTS
END
```

El procedimiento SUMA.CASA hace uso del subprocedimiento SUMLIST a modo de operador. SUMLIST admite una entrada y proporciona una salida; este tipo de procedimientos, que pueden actuar indistintamente como comandos u operadores, reciben el nombre de *procedimientos con parámetros.*

## Procedimientos con parámetros

Al igual que ocurre, en el caso general, con los datos del LOGO cabe distinguir dos tipos de parámetros: de entrada y de salida. Los primeros constituyen datos de entrada al procedimiento y deben especificarse detrás del propio nombre del procedimiento. Por ejemplo:

```
TO SUMLIST :L
```

indica que el procedimiento SUMLIST precisa de un dato de entrada representado por la variable :L. En el mismo caso, el dato en cuestión será identificado, dentro del procedimiento, por :L.

El procedimiento SUMLIST podría definirse de la siguiente forma:

```
TO SUMLIST :L
IF EMPTYP :L [OUTPUT O STOP]
OUTPUT SUM (FIRST :L) (SUMLIST
(BUTFIRST :L))
END
```

Este procedimiento es algo más complicado; si bien, por el momento sólo se prestará atención a los parámetros de entrada y salida.

El parámetro de entrada se ha utilizado como dato en los cálculos. Su empleo es idéntico al de las variables; ello permite que el procedimiento admita distintos datos como entrada.

Respecto a la salida, hay que mencionar al operador OUTPUT (salida). OUTPUT coloca su dato de entrada como dato de salida del procedimiento en el que se encuentra. En el ejemplo propuesto, el mencionado operador se utiliza para «extraer» el valor de la suma. Por supuesto, un determinado procedimiento puede carecer de uno o de ambos tipos de parámetros.

A continuación aparecen dos ejemplos prácticos que ilustran la actuación del operador OUTPUT y resumen el comportamiento de un procedimiento con parámetros.

El primero de ellos adjudica el dato numérico 3 al procedimiento TRES; una operación análoga a las asignaciones de variables, tal como se observa al ordenar la impresión de TRES:

```
TO TRES
OUTPUT 3
END
TRES DEFINED

PRINT TRES
3
```

El segundo ejemplo, algo más evolucionado, ilustra la actuación de un procedimiento con parámetros. En primer lugar, se procede a la definición del procedimiento INICIAL, asociándole un parámetro :PALABRA y se define la función de imprimir el primer elemento del citado parámetro.

```
TO INICIAL :PALABRA
PRINT FIRST :PALABRA
END

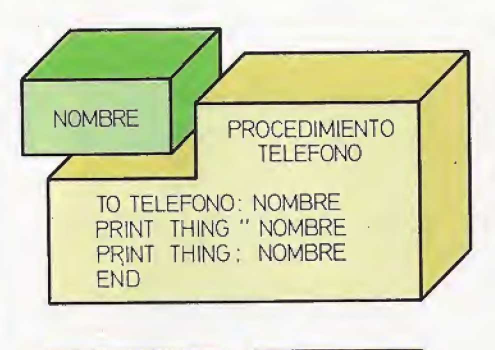
INICIAL DEFINED
```

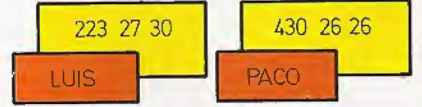
Una vez definido el procedimiento podemos ya utilizarlo en la práctica, sin más que recurrir a su nombre como si se tratara de una orden LOGO.

```
INICIAL "DAVID
D

MAKE "JEFE "PACO
INICIAL :JEFE
P
```

En el primer caso, se le otorga como dato de entrada la palabra DAVID. Su ejecución mostrará en pantalla el primer elemento (letra D) del dato entregado como parámetro.




*La función del operador THING es, en general, devolver el contenido. El procedimiento reflejado en la figura, asociado a las variables LUIS y PACO —cuyos contenidos son los respectivos números de teléfono—, ilustra su doble funcionalidad.*

La entrada asociada al procedimiento con parámetro INICIAL puede ser también una variable. La asignación MAKE, otorga la palabra PACO a la variable JEFE. Ahora, al ejecutar el procedimiento INICIAL sobre el parámetro :JEFE, la pantalla mostrará la primera letra del nombre PACO contenido en la variable JEFE.

## Edición de procedimientos

Si se desea modificar un procedimiento no hace falta escribirlo de nuevo. El comando EDIT facilita esta tarea, visualizando y permitiendo la modificación de procedimientos. EDIT admite un dato de entrada que puede coincidir con un nombre de procedimiento (palabra) o con varios nombres (lista).

Realmente, EDIT abre el acceso al *editor de procedimientos.* Una vez en el editor, se visualiza el procedimiento o procedimientos especificados y el cursor. Este último puede posicionarse allí donde el usuario desee realizar el cambio.

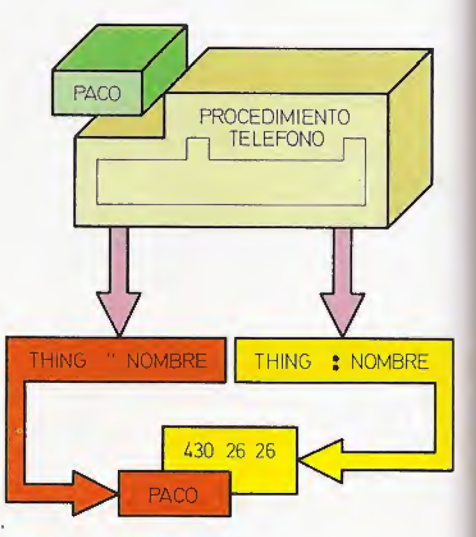
Es posible editar más de un procedimiento a la vez. Para ello es preciso confeccionar una lista con los nombres de los procedimientos y situarla como dato de entrada al comando EDIT. Por ejemplo:

```
EDIT "TRES
EDIT [GASTOS SUMA.TOTAL SUMA.CASA SUMLIST]
```

son formulaciones correctas del mencionado comando. En el primer caso facilitará la modificación del procedimiento cuyo nombre es TRES, mientras que en el segundo permitirá la edición de los cuatro procedimientos cuyos nombres figuran en la lista.

## Verdad y falsedad en LOGO

Además de las posibilidades de manipular palabras y listas, el lenguaje LOGO permite realizar comparaciones e identificaciones de tipo. Los operadores integrados en este grupo, responden




*Al ejecutarse el procedimiento "teléfono", cuyo parámetro de entrada es "nombre", se observa que el operador THING puede devolver tanto el nombre de la variable introducida como parámetro (THING "NOMBRE), como el contenido de la misma (THING :NOMBRE).*

con TRUE (cierto) o FALSE (falso) como datos de salida.

Un primer operador al respecto es WORDP. Este responderá TRUE en el caso de recibir como entrada una palabra y FALSE en caso contrario.

De forma análoga actúan LISTP y NUMBERP con listas y números, respectivamente.

Un buen ejercicio es combinar estos dos operadores con los estudiados en anteriores capítulos, para identificar el tipo de las salidas que entregan en cada situación. Por ejemplo:

```
PRINT WORDP [CURSO DE BASIC]
FALSE

PRINT WORDP FIRST [CURSO DE BASIC]
TRUE

PRINT LISTP BUTLAST LIST "DON :YO
TRUE
```

Tanto en el ámbito de las palabras como de las listas, cabe la nulidad; esto es: la palabra vacía (") o la lista vacía ([]) en las que no interviene ningún elemento. El operador EMPTYP evalúa tal situación, y responde con TRUE si su dato de entrada coincide con una palabra o una lista vacía. Hay que tener en cuenta que para el LOGO una palabra vacía y una lista vacía no son la misma cosa. Ello puede comprobarse utilizando el operador EQUALP.

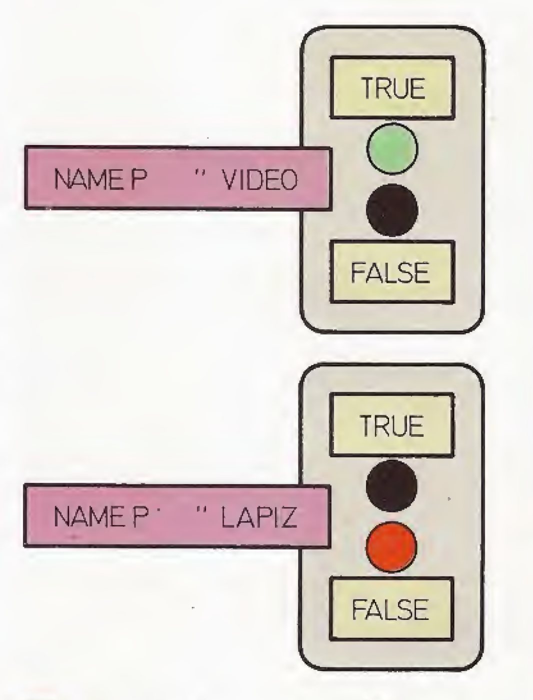
EQUALP compara dos entradas y comunica si son o no iguales.

```
PRINT EMPTY []
TRUE

PRINT EQUALP" []
FALSE

PRINT EQUALP "CASA [CASA]
FALSE
```

*Para evaluar el tipo de dato manipulado, el LOGO cuenta con tres operadores adecuados para detectar palabras, listas y números. Estos son, respectivamente, WORDP, LISTP y NUMBERP.*







*La función del operador NAMEP es averiguar si una palabra coincide con el nombre de una variable previamente definida.*

Un último operador de esta categoría es MEMBERP. Necesita dos datos de entrada: el primero puede ser una palabra, un número o una lista, mientras que el segundo debe ser obligatoriamente una lista.

La respuesta será TRUE si la primera entrada es un elemento que forma parte de lista indicada. Es evidente que su utilidad se manifiesta a la hora de evaluar la presencia de una palabra o de una sublista dentro de una lista más compleja. Los siguientes ejemplos, basados en la LISTA3 definida, revelan su actuación.

```
PRINT MEMBERP "EL :LISTA3
MAKE "LISTA3 [EL LENGUAJE LOGO]
TRUE

PRINT MEMBERP [EL] :LISTA3
FALSE

PRINT MEMBERP 7[1[3]5 [7]9]
FALSE
```

## Algo más sobre variables

Por el momento, el trabajo con variables se ha centrado casi por completo en el uso del comando MAKE, adecuado para asignar un dato a la variable especificada. Las posibilidades de MAKE no terminan ahí. Cada vez que se ejecuta el comando MAKE acompañado de un nuevo nombre de variable, éste reserva espacio para esa variable.

Al respecto, cabe recordar que no es posible utilizar una variable sin antes definirla previamente por medio de MAKE.

El LOGO lleva buena cuenta de las variables definidas.

Un método para averiguar si una palabra coincide con el nombre de una variable lo aporta el operador NAMEP.

Este admite un dato de entrada y responde con TRUE si el mencionado dato es el nombre de una variable previamente definida. Al igual que otros operadores similares, descritos anteriormente, su verdadera utilidad práctica saltará a la luz al abordar el estudio de los bucles.

## El operador THING

En algunos casos, es conveniente que el contenido de una variable coincida con el nombre de otra. Esta es una situación que queda ilustrada por el siguiente ejemplo.

**TABLA DE ORDENES-LOGO (1)**

| Instrucción | Cometido | Operador/Comando |
|---|---|---|
| COUNT <objeto> | Cuenta elementos | Operador |
| FIRST <objeto> | Da el primer elemento | Operador |
| LAST <objeto> | Da el último elemento | Operador |
| BUTFIRST <objeto> | Elimina primer elemento | Operador |
| BUTLAST <objeto> | Elimina último elemento | Operador |
| LPUT | Añade al final | Operador |
| FPUT | Añade al principio | Operador |
| LIST | Construye una lista | Operador |
| SE | Construye una lista de palabras | Operador |
| WORD | Construye palabras | Operador |
| ASCII <palabra> | Da el código ASCII de la inicial | Operador |
| CHAR <número> | Da el carácter ASCII correspondiente | Operador |

<objeto>: puede ser una palabra, una lista, un número o una variable (ver tabla de entradas y salidas).

**NATURALEZA DE LOS DATOS DE ENTRADA Y SALIDA**

| Operador | Entrada 1 | Entrada 2 | Salida |
|---|---|---|---|
| COUNT | <pal/lis> | | <número> |
| FIRST/LAST | <palabra> | | <carácter> |
| | <lista> | | <palabra/sublista> |
| BUTFIRST/BUTLAST | <palabra> | | <palabra> |
| | <lista> | | <lista> |
| LPUT/FPUT | <pal/lis> | <lista> | <lista> |
| WORD | <palabra> | <palabra> | <palabra> |
| LIST | <pal/lis> | <pal/lis> | <lista> |
| SE | <pal/lis> | <pal/lis> | <lista> |
| ASCII | <carácter> | | <número> |
| CHAR | <número> | | <carácter> |

**TABLA DE ORDENES-LOGO (2)**

| Instrucción | Cometido | Operador/Comando |
|---|---|---|
| MAKE <palabra> <dato> | Crea variable | Comando |
| TO <nombre> [<variables>] | Crea procedimiento | Comando |
| END | Determina el fin de un procedimiento | Comando |
| EDIT [<nombre>...] | Activa el editor de procedimientos | Comando |

<nombre>: nombre del procedimiento (sin comillas).
<variables>: posibles variables locales (precedidas por dos puntos).

Una vez introducidas las dos instrucciones que siguen:

```
MAKE "NUMERO "DIEZ
MAKE "DIEZ 10
```

para acceder al número 10 es preciso recurrir al nombre de variable :DIEZ. Esta es una función realizable por medio de un nuevo operador: THING.

THING "DIEZ actúa exactamente igual que :DIEZ, accediendo al número 10: THING admite como entrada la palabra que constituye el nombre de una variable y devuelve su contenido. No obstante, THING permite algo más.

En el mismo ejemplo, la salida de THING "NUMERO será "DIEZ; sin embargo, la salida de THING :NUMERO, coincidirá con 10. En definitiva, con THING es posible acceder al «contenido del contenido» de una variable.

Un ejemplo esclarecedor lo aporta el siguiente programa. Empieza con la asignación de dos variables (dos nombres a los que se asigna su correspondiente número de teléfono). A continuación, se define un procedimiento al que se asocia el parámetro :NOMBRE; un procedimiento que incluye la doble posibilidad de uso del operador THING: acompañado por un nombre de variable expresado como palabra ("NOMBRE) o como variable (:NOMBRE).

```
MAKE "LUIS 123 28 30
MAKE "PACO 430 26 26
TO TELEFONO :NOMBRE
PRINT THING "NOMBRE
PRINT THING :NOMBRE
END
TELEFONO DEFINED
```

La ejecución evidencia las dos posibles actuaciones del operador THING. Al ejecutar el procedimiento TELEFONO, dando como parámetro un nombre de variable, el ordenador responderá presentando el nombre en cuestión (PRINT THING "NOMBRE), seguido por el número de teléfono o contenido de la mencionada variable (PRINT THING :NOMBRE).

TELEFONO "LUIS
*LUIS*
*123 28 30*

TELEFONO "PACO
*PACO*
*430 26 26*

Obsérvese que THING "NOMBRE equivale a :NOMBRE y que THING :NOMBRE equivale a THING THING "NOMBRE.

## Variables globales y locales

Al hablar de procedimientos con parámetros se indicó que los parámetros de entrada actúan como variables asociadas a la ejecución del referido procedimiento. Estas variables son de un tipo especial y reciben el nombre de variables locales.

Las variables locales se diferencian de las restantes en varios aspectos.

— No es preciso definirlas por medio del operador MAKE.

— Su nombre no es identificado por NAMEP.

— Sólo tienen validez dentro del procedimiento en el que se encuentran.

Las variables globales pueden ser visualizadas y modificadas por medio del *editor de variables,* al que se accede a través del comando EDNS (EDit NameS). El editor de variables funciona del mismo modo que el de procedimientos, con la salvedad de que también permite actualizar el valor de las variables globales.

## Espacio de trabajo

El *espacio de trabajo* (Work Space) es el conjunto de procedimientos y variables, definidos por el usuario, que se encuentran en la memoria del ordenador.

El lenguaje LOGO dispone de varios comandos para el manejo del espacio de trabajo.

En primer lugar están los comandos que muestran el contenido actual del espacio de trabajo. Todos ellos comienzan por PO (Print Out):

POTS (Print Out TitleS): muestra todos los nombres (títulos) de los procedimientos existentes en el espacio de trabajo.

POPS (Print Out ProcedureS): visualiza en la pantalla el contenido de todos los procedimientos.

Si lo que se desea es visualizar exclusivamente el contenido de un determinado procedimiento, hay que utilizar el prefijo PO seguido por el nombre del procedimiento. Por ejemplo:

```
PO "INICIAL
```

mostrará en la pantalla el procedimiento INICIAL.

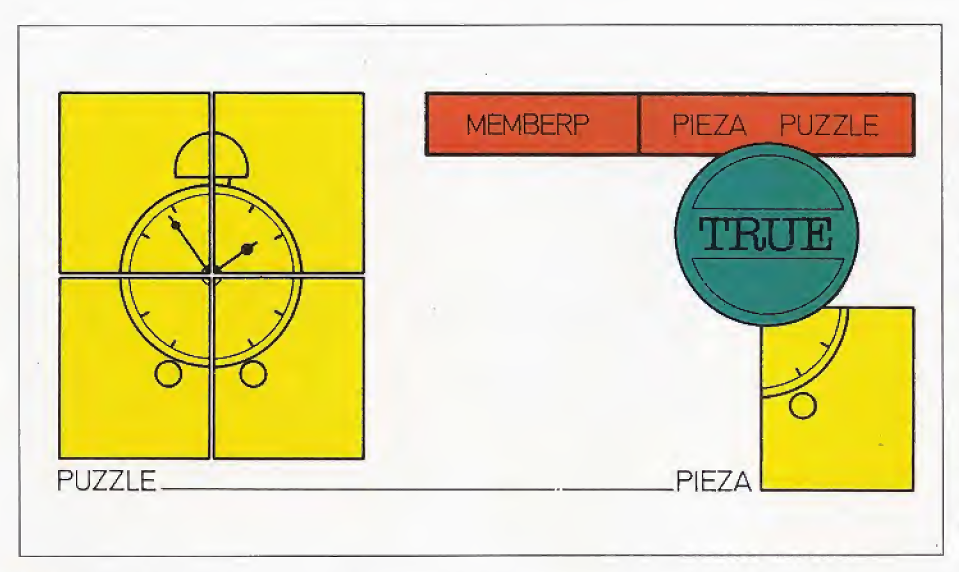
En general, los referidos comandos permiten ver, pero no modificar los procedimientos; para efectuar cualquier cambio es preciso recurrir al comando para la edición de procedimientos: EDIT.

Un segundo grupo de comandos especializados en el tratamiento del espacio de trabajo, está integrado por los destinados a visualizar las variables globales definidas.

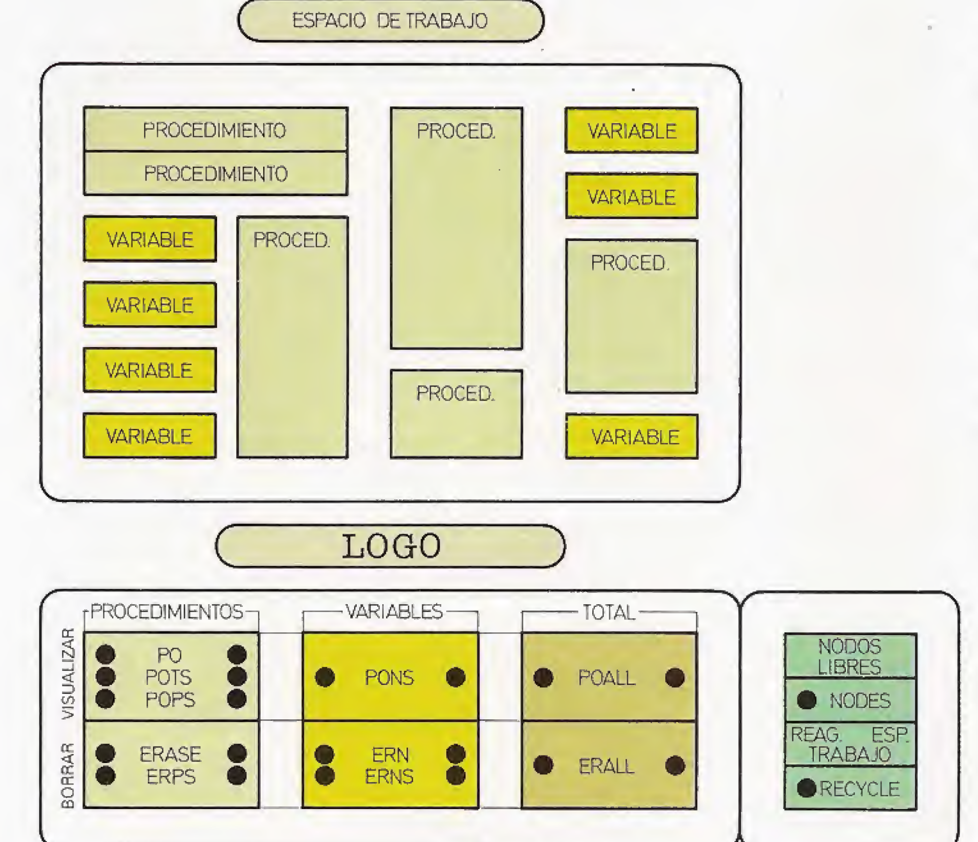
PONS (Print Out NameS): muestra los nombres y contenidos de las variables; no obstante, al igual que PO y POPS, no permite su modificación.

| ORDENES DE CONTROL DEL ESPACIO DE TRABAJO | | |
|---|---|---|
| **Instrucciones** | **Cometido** | **Operador/Comando** |
| ERALL | Borra el espacio de trabajo | Comando |
| ERASE <objeto> | Borra procedimiento(s) | Comando |
| ERN <objeto> | Borra variable(s) | Comando |
| ERNS | Borra todas las variables | Comando |
| ERPS | Borra todos los procedimientos | Comando |
| PO <objeto> | Muestra procedimiento | Comando |
| POALL | Muestra espacio de trabajo | Comando |
| PONS | Muestra variables (globales) | Comando |
| POPS | Muestra procedimientos (definiciones) | Comando |
| POTS | Muestra procedimientos (títulos) | Comando |
| NODES | Da el número de nodos libres | Operador |
| RECYCLE | Reorganiza memoria | Comando |
| <objeto> puede ser una palabra o una lista | | |

| TABLA DE ORDENES-LOGO (3) | | |
|---|---|---|
| **Instrucción** | **Cometido** | **Operador/Comando** |
| WORD <objeto> | TRUE si <objeto> es palabra | Operador |
| LISTP <objeto> | TRUE si <objeto> es una lista | Operador |
| NUMBERP <objeto> | TRUE si <objeto> es un número | Operador |
| EMPTY <objeto> | TRUE si <objeto> vacío | Operador |
| EQUALP <obj1> <obj2> | TRUE si <obj1> y <obj2> son iguales | Operador |
| MEMBERP <obj> <lista> | TRUE si <obj> es elemento de <lista> | Operador |
| NAMEP <palabra> | TRUE si es nombre de variable | Operador |
| THING <objeto> | Devuelve el contenido | Operador |
| EDNS | Activa el editor de variables | Comando |
| <objeto> puede ser una palabra o una variable. <nombre> es el nombre del procedimiento (sin comillas). <variables> son las posibles variables locales (con dos puntos). | | |




*MEMBERP es otro de los operadores LOGO cuya respuesta es TRUE (cierto) o FALSE (falso). Se ejecución permite confirmar la presencia de una palabra o de una sublista dentro de la lista especificada.*




*Ordenes LOGO para el control del «espacio de trabajo» o conjunto de procedimientos y variables, definidos por el usuario, que se encuentran en la memoria del ordenador.*

POALL (Print Out ALL): este es otro comando afecto al espacio de trabajo; su especialidad es presentar en la pantalla el contenido de todo el espacio de trabajo. Existe también la posibilidad de borrar una parte o la totalidad del espacio de trabajo. Los comandos utilizables para tal fin comienzan por ER (ERase).

ERASE: borra el procedimiento o procedimientos cuyos nombres acompañan al comando.

Por ejemplo:

ERASE "CASA

hace que desaparezca del espacio de trabajo el procedimiento CASA.

Tal como se ha indicado, ERASE admite también una lista de nombres de procedimientos; todos ellos serán borrados por efecto del mismo comando.

ERPS (ERase ProcedureS): éste es un comando LOGO cuya especialidad es la de borrar, a la vez, todos los procedimientos que se encuentran en el espacio de trabajo.

Por lo que respecta a las variables, es posible borrar sólo algunas o todas a la vez. ERN borra la variable o lista de variables especificada, mientras que ERNS (ERase NameS) borra todas las variables de «un plumazo». Hay que señalar que ambos comandos eliminan tanto el contenido como el nombre de las variables, con lo cual, las variables borradas desaparecen por completo del espacio de trabajo. Por último, cabe mencionar al comando ERALL (ERase All), cuya ejecución borra todo el espacio de trabajo.

Cada vez que se crea un procedimiento o variable queda ocupada una zona del espacio de trabajo. Para conocer cuánto espacio queda aún disponible se utiliza el operador NODES. Este responde con el número de «nodos» libres. Cada nodo equivale a cinco bytes de memoria.

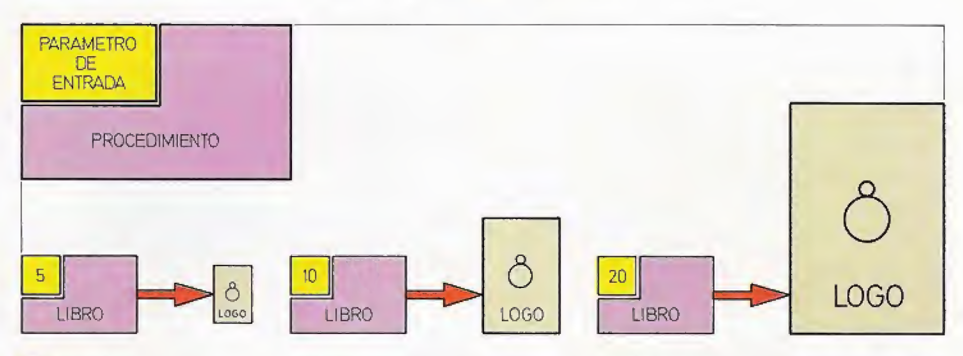
Al eliminar procedimientos o variables se producen «huecos» en el espacio de trabajo ocupado. Para reordenar los procedimientos y variables hay que recurrir al comando RECYCLE. RECYCLE reúne los nodos libres reagrupando el espacio de trabajo utilizado. NODES proporciona el número real de nodos libres si se utiliza previamente RECYCLE. Un dato a señalar es que si no queda espacio suficiente a la hora de crear un procedimiento, se ejecuta automáticamente el comando RECYCLE.

# LOGO (3)

## TURTLE GRAPHICS: Los gráficos del LOGO



n un capítulo anterior ya se presentaron los fundamentos del tratamiento de gráficos con el LOGO. Asimismo, en el capítulo anterior se han detallado el concepto y uso de los procedimientos. Este capítulo va a combinar ambos temas, trasladando las posibilidades que ofrecen los procedimientos al terreno de la confeción de dibujos con la tortuga. Se crearán procedimientos capaces de trazar gráficos y se verán las enormes posibilidades que éstos aportan a la confección de gráficos.




*La técnica de los procedimientos LOGO es íntegramente aplicable al método «Turtle Graphics». Las secuencias de órdenes de dibujo pueden adoptar la forma de procedimientos, con o sin parámetros.*

### Procedimientos con la tortuga

Uno de los ejemplos incluidos en la presentación del «Turtle graphics» fue la secuencia de órdenes adecuadas para trazar una línea en la zona superior de la pantalla, devolviendo a la tortuga, a continuación, a su posición original. El referido conjunto de órdenes puede integrarse dentro de un procedimiento, sin más que apelar al método de definición ya estudiado; por ejemplo:

```
TO LINEA
CS
PENUP
FORWARD 50
RIGHT 90
PENDOWN
FORWARD 50
PENUP
HOME
END
```

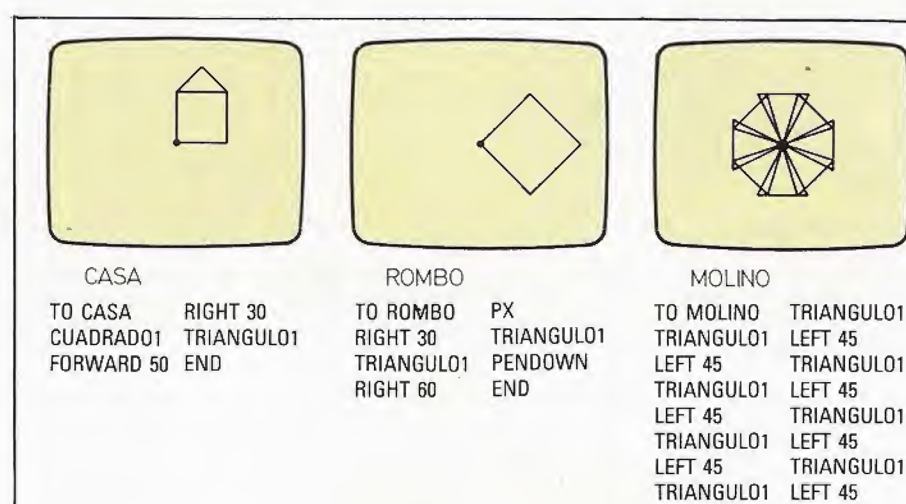
Al integrar la secuencia de órdenes dentro de un procedimiento, se evita la necesidad de teclear de nuevo las mismas órdenes cada vez que desee realizarse la misma acción.

Todas las posibilidades de la creación de gráficos por medio de la tortuga pueden incluirse en procedimientos. A continuación, se describen algunos procedimientos con la tortuga que pueden resultar útiles a la hora de trazar dibujos más complejos.

El primero de ellos, cuyo nombre es CUADRADO 1, es capaz de dibujar un cuadrado de 50 posiciones de lado. Tras confeccionar el dibujo, la tortuga queda emplazada en su posición de origen, tal como muestra la correspondiente pantalla.

```
TO CUADRADO1
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
END
```

Un nuevo ejemplo lo aporta el procedimiento que sigue (TRIANGULO1); su ejecución dibuja un triángulo cuyo lado es de 50 posiciones.




*La aplicación de la técnica de los procedimientos LOGO al dibujo con la tortuga, permite construir procedimientos gráficos, cada vez más complejos, al asociar varios procedimientos elementales. Las pantallas ilustran la ejecución de tres procedimientos en cuya definición intervienen otros más primitivos (CUADRADO1 y TRIANGULO1) definidos con anterioridad.*

```
TO TRIANGULO1
FORWARD 50
RIGHT 120
FORWARD 50
RIGHT 120
FORWARD 50
RIGHT 120
END
```

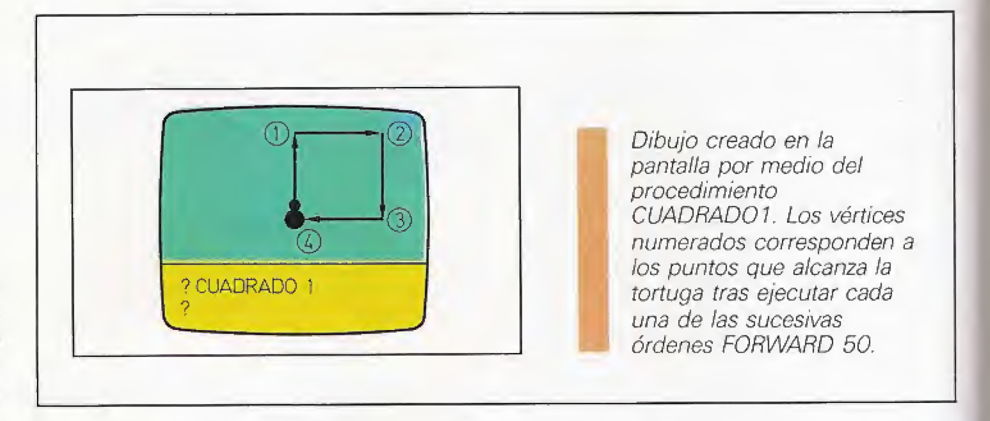
De nuevo, en este segundo ejemplo, una vez realizado el dibujo, la tortuga queda en la posición de partida. Cabe notar que la suma de los ángulos girados es de 360 grados (una vuelta completa), de ahí que el retorno de la tortuga a la posición de origen, y con la orientación adecuada, sea directo.

Ahora, es posible utilizar los dos procedimientos elementales definidos para crear figuras más complicadas. Un ejemplo lo constituye el procedimiento BANDERIN1 que hace uso de TRIANGULO1 para dibujar un banderín con mástil.

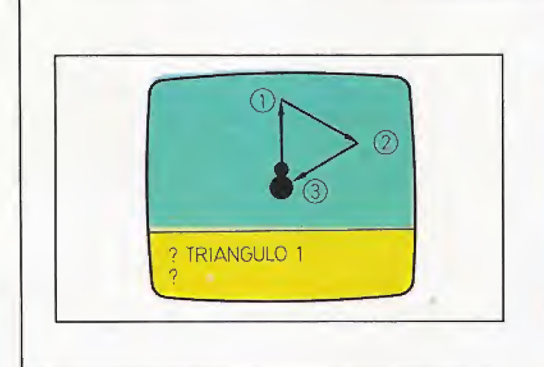
```
TO BANDERIN1
FORWARD 30
TRIANGULO1
END
```

El cuadro adjunto contiene tres nuevos ejemplos de procedimientos de dibujo más evolucionados (CASA, ROMBO y MOLINO), que hacen uso de los ya definidos: CUADRADO1 y TRIANGULO1.

Estas son algunas posibilidades que brinda la aplicación de los procedimientos LOGO al método «Turtle graphics». Por supuesto, la versatilidad de los procedimientos con parámetros también es trasladable al caso de la tortuga. Muestra de ello van a constituirla los ejemplos de los próximos apartados.




*Dibujo creado en la pantalla por medio del procedimiento CUADRADO1. Los vértices numerados corresponden a los puntos que alcanza la tortuga tras ejecutar cada una de las sucesivas órdenes FORWARD 50.*




*El procedimiento TRIANGULO1 instruye a la tortuga para que trace en la pantalla el dibujo ilustrado. La ejecución de cada una de las tres órdenes FORWARD que en él intervienen conducen a la tortuga a los distintos vértices del triángulo.*

## Procedimientos a medida

Una vez creados los procedimientos CUADRADO1 y TRIANGULO1 quedan definidos ambos dibujos en el campo de trabajo. Para utilizarlos, no hay más que llamar al respectivo procedimiento utilizando su nombre. Las referidas figuras pueden trasladarse a cualquier posición de la pantalla. Para ello basta con colocar la tortuga en la posición y orientación adecuadas. E incluso puede alterarse el tamaño de los dibujos. Esto último exige algunos retoques en el procedimiento para que sea posible introducir la longitud de un lado. Tal valor puede introducirse como parámetro de entrada al procedimiento. Veamos cuál es el aspecto de ambos ejemplos una vez convertidos en procedimientos con un parámetro de entrada.

```
TO CUADRADO :L
FORWARD :L
RIGHT 90
FORWARD :L
RIGHT 90
FORWARD :L
RIGHT 90
FORWARD :L
RIGHT 90
END
```

En el primer caso (dibujo de un cuadrado), la diferencia aparece en el parámetro L. Su cometido es definir la magnitud del lado del cuadrado, de ahí que su valor acompañe a las cuatro órdenes FORWARD.

Análogamente, puede definirse un procedimiento capaz de dibujar un triángulo cuyo lado lo precisará el parámetro de entrada L.

```
TO TRIANGULO :L
FORWARD :L
RIGHT 120
FORWARD :L
RIGHT 120
FORWARD :L
RIGHT 120
END
```

En definitiva, los dos nuevos procedimientos permiten definir el tamaño de la figura a través del valor o parámetro que se agrega como entrada al procedimiento. El siguiente ejemplo, CUADROS, utiliza esta posibilidad para dibujar en la pantalla un conjunto de cuadrados de distinto lado.

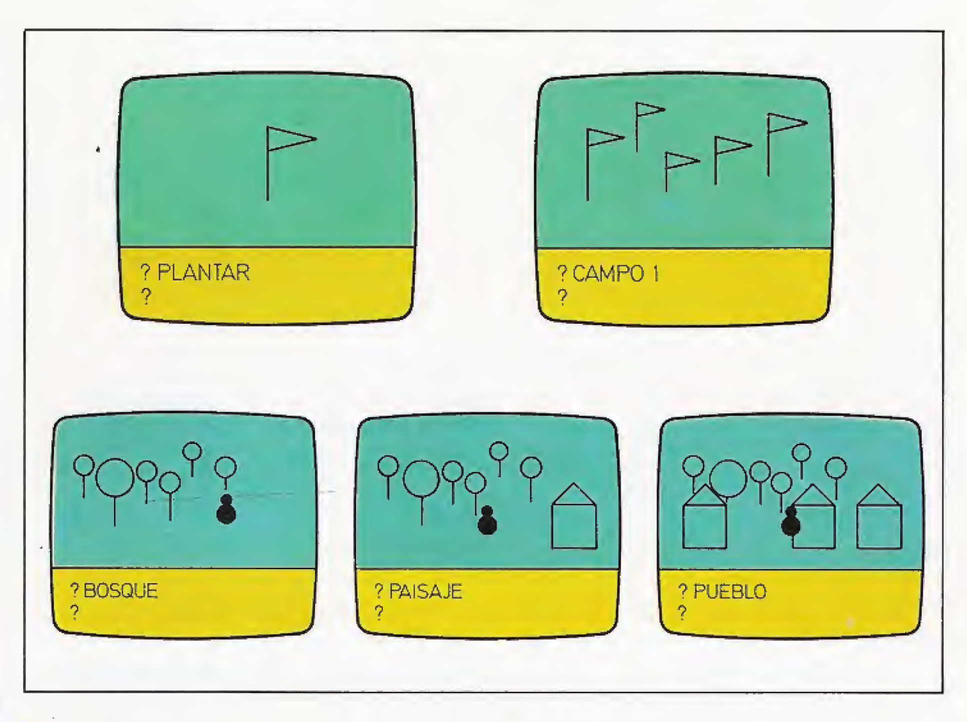
CUADROS no es más que un procedimiento evolucionado, creado a partir de la repetición del procedimiento con parámetro CUADRADO :L.

```
TO CUADROS
CUADRADO 20
CUADRADO 40
CUADRADO 60
CUADRADO 80
CUADRADO 100
END
```

De la misma forma se pueden definir procedimientos más genéricos. Por ejemplo, uno que dibuje polígonos regulares dando el número de lados y su longitud; tal es el caso del procedimiento POLIGONO definido a continuación:

```
TO POLIGONO :N :L
REPEAT :N [FORWARD :L RIGHT 360/:N]
END
```

Su definición, extremadamente compacta, adopta el aspecto de procedimiento con dos parámetros de entrada: N (número de lados del polígono) y L



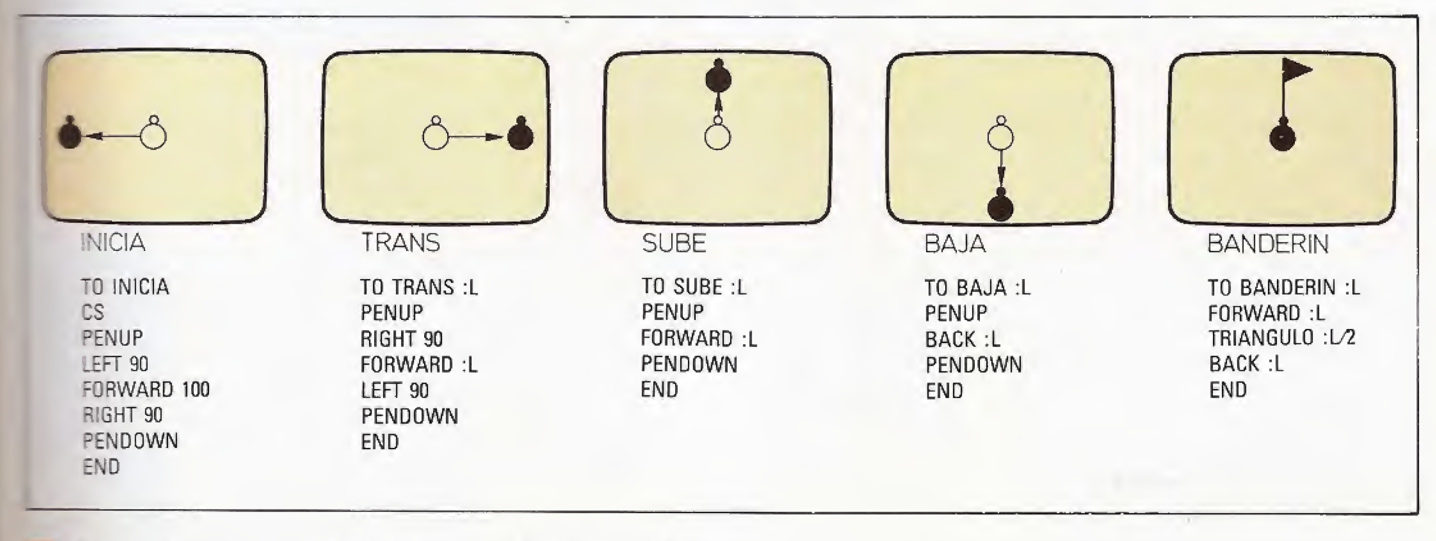

*Resultado de la ejecución de dos procedimientos de dibujo definidos en el texto: PLANTAR y CAMPO1.*

(longitud del lado). Un aspecto a señalar es la presencia dentro del procedimiento del comando REPEAT. Su estudio detallado tendrá lugar en el capítulo cuatro de la serie dedicada al LOGO; por el momento basta con saber que ordenará la repetición de las acciones encerradas entre corchetes tantas veces como indique el parámetro N.

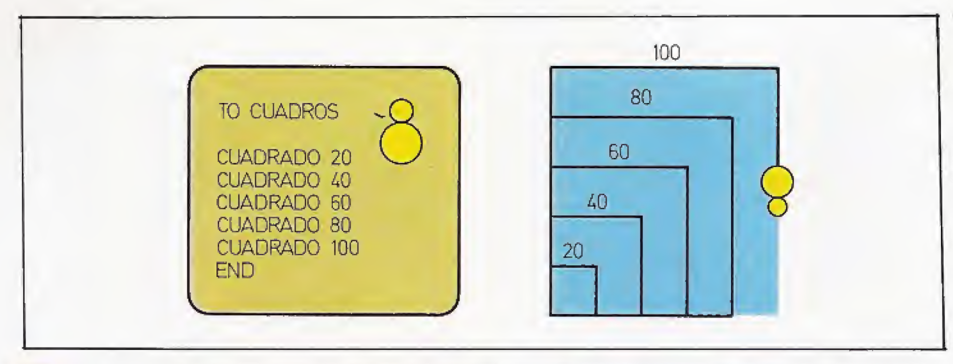
Si el número de lados especificado es 3 ó 4, POLIGONO se comporta como los procedimientos TRIANGULO o CUADRADO, respectivamente.

## Colección de ejemplos

Para trasladar al terreno práctico los conocimientos adquiridos hasta el mo-




*Cinco procedimientos elementales o «de utilidad», adecuados para la creación de superprocedimientos gráficos.*




*Ejecución del superprocedimiento CUADROS. Dentro de su definición interviene repetidamente un procedimiento (CUADRADO) cuyo parámetro de entrada coincide con el lado del cuadrado a dibujar.*

mento, vamos a poner a trabajar a la disciplinada tortuga.

La siguiente colección de ejemplos hará uso de los procedimientos elementales o de utilidad INICIA, TRANS, SUBE, BAJA y BANDERIN que se detallan en el cuadro correspondiente. Los referidos procedimientos de utilidad entrarán a formar parte de la definición de superprocedimientos más evolucionados, capaces de «instruir» a la tortuga para que trace en la pantalla dibujos más elaborados.

Al utilizar los cinco procedimientos elementales, definidos en el cuadro adjunto, nuestro simpático dibujante —la tortuga— debe estar orientada en sentido vertical y «mirando» hacia el borde superior de la pantalla.

Con esta precaución, la actuación de los referidos procedimientos elementales será la que sigue:

INICIA: desplaza a la tortuga a la izquierda de la pantalla, levantando la tiza para que no deje trazo al trasladar su origen.

TRANS: traslada a la tortuga tantas posiciones a la derecha como indique el valor del parámetro L; de nuevo, sin dejar trazo alguno en la pantalla.

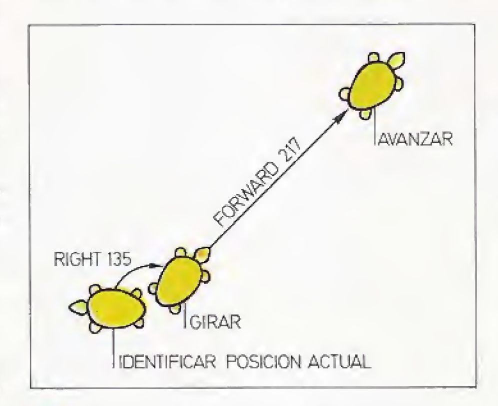
SUBE, BAJA: ambos procedimientos permiten alterar la posición vertical de la tortuga: hacia arriba o hacia abajo, respectivamente.

BANDERIN: dibuja un banderín del tamaño especificado.

Con estos procedimientos elementales, van a dibujarse a continuación varias escenas, sencillas aunque plenamente ilustrativas de las posibilidades que otorgan los procedimientos al trazado de gráficos en pantalla. Desde luego, el usuario puede crear su propios dibujos utilizando y alterando alguno de los siguientes ejemplos. El primero de los ejemplos (PLANTAR), hace uso de los procedimientos SUBE, BANDERIN y BAJA para dibujar un banderín en la zona superior de la pantalla.

```
TO PLANTAR :L
SUBE :L
BANDERIN 100–(:L/3)
BAJA :L
END
```

A su vez, CAMPO1, es un procedimiento evolucionado que recurre a los procedimientos elementales INICIA, BANDERIN, TRANS y PLANTAR para dibujar en la pantalla una escena con cinco banderines:




*Secuencia de pasos necesarios para situar a la tortuga en un determinado punto de la pantalla.*

```
TO CAMPO1
INICIA
BANDERIN 60
TRANS 50
PLANTAR 60
TRANS 40
PLANTAR 20
TRANS 60
PLANTAR 35
TRANS 40
PLANTAR 53
END
```

A continuación, se desarrollan nuevos ejemplos a partir de POLIGONO. El procedimiento CIRCUNFERENCIA particulariza el procedimiento POLIGONO, definido en el apartado anterior, para un número grande de lados (30) y calcula su longitud partiendo del radio. Este procedimiento se utiliza dentro de ARBOL para dibujar la copa.

```
TO ARBOL :L
FORWARD :L
LEFT 90
CIRCUNFERENCIA :L*3/4
RIGHT 90
BACK :L
END
```

```
TO CIRCUNFERENCIA :R
POLIGONO 30 ((:R*6.28)/30)
end
```

A su vez, BOSQUE se apoya en el procedimiento ARBOL, situando cada elemento en la pantalla con la ayuda del procedimiento PON.

```
TO PON :L
SUBE :L
ARBOL (100-:L)/3
BAJA :L
END
```

Este último es similar a PLANTAR,

aunque recurriendo a ARBOL en lugar de a BANDERIN. Su empleo repetido dentro del procedimiento BOSQUE da lugar a la representación gráfica que muestra la correspondiente pantalla.

```
TO BOSQUE
INICIA
PON 50
TRANS 40
PON 20
TRANS 40
PON 35
TRANS 40
PON 17
TRANS 40
PON 65
TRANS 40
PON 44
END
```

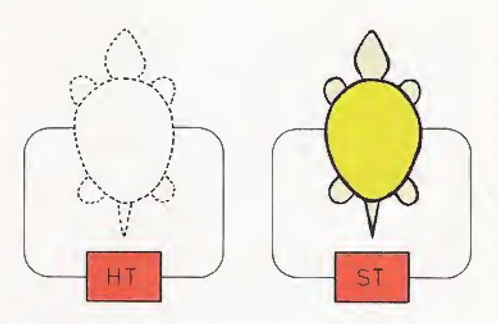
Cada vez los dibujos pueden ser más y más complejos, al irse enriqueciendo con la incorporación de procedimientos definidos a base de otros más elementales.

Por ejemplo, introduciendo algunas variantes al procedimiento INICIA puede definirse un nuevo procedimiento que denominaremos INI; esta vez, el desplazamiento de la tortuga no borrará el contenido de la pantalla.

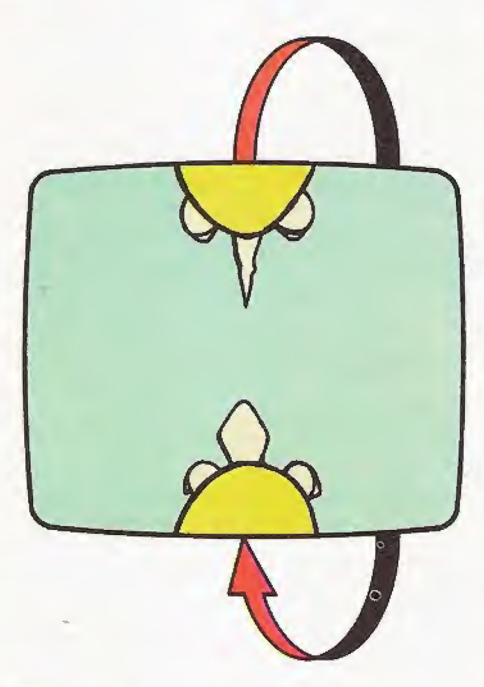
```
TO INI
PENUP
HOME
LEFT 90
FORWARD 120
RIGHT 90
PENDOWN
END
```

Asociando los procedimientos BOSQUE, BAJA y CASA es posible ya el dibujo de un paisaje en el que aprezcan árboles e incluso una casa.

```
TO PAISAJE
BOSQUE
BAJA 30
CASA
PENUP
HOME
PENDOWN
END
```




*El comando HT hace que la tortuga desaparezca de la pantalla. A pesar de ello, ésta puede seguir trazando dibujos que serán visibles. Para restaurar su presencia, es preciso utilizar el comando ST.*



*Al operar en modo cerrado, seleccionado con el comando WRAP, la tortuga reaparecerá en la pantalla por el extremo opuesto al que ha desaparecido.*

La evolución puede seguir en sucesivas etapas. Por ejemplo, introduciendo el procedimiento anterior (PAISAJE) dentro de un nuevo superprocedimiento capaz de plasmar en la pantalla el dibujo de un pueblo de la mano de la tortuga LOGO.

```
TO PUEBLO
PAISAJE
BAJA 40
CASA
INI
BAJA 50
CASA
PENUP
HOME
PENDOWN
END
```

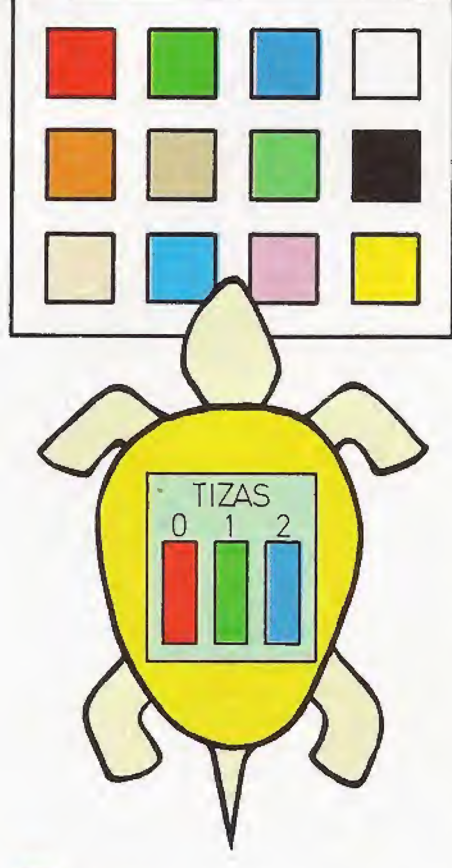
## La tortuga se esconde

Una vez terminado un dibujo, puede ser conveniente evitar la presencia de la tortuga. Para conseguirlo, puede desplazarse a la tortuga a una esquina de la pantalla. No obstante, el método más eficaz es ordenarla que se esconda. Para ello, se dispone del comando HT (Hide Turtle). Este comando vuelve invisible a la tortuga, aunque no impide su movimiento ni altera el modo de dibujo seleccionado (PD, PE, PU o PX). Si se desea recuperar su presencia, bastará con teclear la orden opuesta ST (Show Turtle).

Existe otro método para perder el contacto visual con tan simpático personaje: obligarlo a traspasar los límites de la pantalla. Hay dos alternativas para fijar dichos límites; éstas coinciden con los denominados *modo abierto* y *modo cerrado.*

En el primero, la pantalla muestra únicamente una parte del campo de acción de la tortuga. si la tortuga sale de esta zona visible o ventana, desaparecerá de la vista del usuario. Este modo se selecciona con el comando WINDOW (ventana).

En el segundo caso, modo cerrado, la tortuga no puede salir de la pantalla. Si intenta evadirse por un extremo, aparecerá de inmediato por el opuesto. Una imagen gráfica de tal situación se obtiene al pensar que la pantalla es una lámina flexible, a la que doblamos y pegamos sus lados opuestos; esto es: el borde izquierdo con el derecho y el superior con el inferior. En tal caso, es obvio que si la tortuga desaparece por el borde superior, reaparecerá de inmediato por el inferior. Y, análogamente, si se ausenta por el lado izquierdo de la pantalla, reaparecerá instantáneamente por el lado derecho. Para optar por este segundo modo se dispone del comando




*Dependiendo de las propias características del ordenador, la tortuga será capaz de utilizar un mayor o menor número de tizas de colores. En todo caso, hay que tener en cuenta que ésta sólo puede llevar un número reducido de tizas. Para que le sea posible utilizar los restantes colores, debe intercambiar las correspondientes tizas que lleve «puestas» en ese preciso instante.*

WRAP (enrollar). Es preciso tener cuidado al usar estos dos comandos pues ambos borran la pantalla.

Es posible que, en determinado momento, no se vea a la tortuga y se ignore si está escondida o se encuentra fuera de la pantalla. En éste y en otros casos semejantes, es aconsejable el uso de SHOWNP. El operador SHOWNP devuelve el valor FALSE si se ha ejecutado una orden HT para obligar al quelonio a desaparecer. Su ejecución permite conocer el estado actual, visible o invisible, activado por los comandos ST o HT, respectivamente. Si lo que ocurre es que la tortuga ha salido de la pantalla, pero no se le ha ordenado esconderse, SHOWNP responderá con TRUE.

## Tizas de colores

Por el momento se han confeccionado dibujos, más o menos complejos, controlando los desplazamientos del personaje central del «turtle graphics», pero siempre a base de trazos de un solo color. Desde luego, los dibujos resultarían más atractivos si fuera posible colorearlos.

En la primera instancia, las posibilidades de elección de color dependen del ordenador que se utilice. Al respecto, es preciso consultar el manual propio del aparato con objeto de aprovechar las distintas opciones que brinda. Una vez precisadas las posibilidades de color del equipo, hay que poner en práctica los medios que incorpora el LOGO para controlar las tizas de color.

En principio, hay que señalar que la tortuga puede transportar más de una tiza. Estas están numeradas para facilitar su identificación.

Para cambiar la tiza se utiliza la orden SETPN, seguida por el número de tiza deseado. Dicho comando permite trazar dibujos que aparecerán en la pantalla con el color seleccionado. La nueva tiza seguirá utilizándose hasta que se cambie de nuevo su color al ejecutar un nuevo comando SETPN. El número habitual de tizas que puede transportar la tortuga se eleva a tres. Por otra parte, en cualquier momento se puede identificar el número de tiza en uso, por medio del operador PN; su ejecución indica cuál es el número de la tiza en acción.

Lo más frecuente es que el ordenador permita utilizar un número de colores superior a tres. Para hacer uno de los restantes colores será necesario cambiar el juego de las tres tizas que puede viajar con la tortuga. El comando que reasigna los colores de las tizas es SETPC. SETPC exige dos entradas: la primera debe coincidir con el número de tizas a reasignar, mientras que la segunda debe ser un número que identifique a uno de los colores que permite utilizar el ordenador.

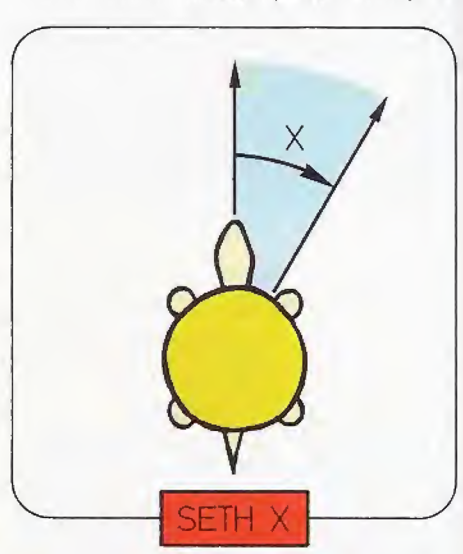
El operador PC constituye una herramienta de ayuda para el usuario en este punto, puesto que al utilizarlo acompañado por un número de tiza, la máquina responderá con el número del color asignado a la misma.

## Cambio del fondo

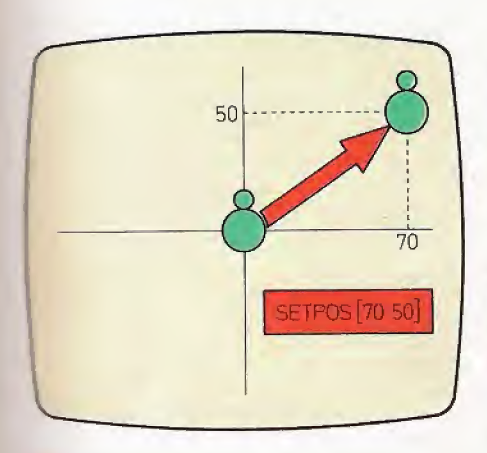
Al emplear tizas de colores puede ocurrir, en algún caso, que se esté pintando con una tiza cuyo color coincide con el color del fondo. De darse esta situación, los dibujos no resultarán visibles. Para solventar esta situación, el LOGO permite alterar el color del fondo de la pantalla.

El comando adecuado es SETBG (SET-BackGround). Al igual que ocurre con las tizas, los colores disponibles dependerán de la gama que ofrezca el ordenador. SETBG debe ir seguido por el correspondiente número de color. Esta posibilidad permite alterar con comodidad el escenario del dibujo: verde para un paisaje campestre, azul para un fondo marino o negro para una escena galáctica.

De nuevo, el usuario puede averiguar




*La orientación definida por el comando SETH afecta a la tortuga en términos absolutos. Esto es, la tortuga girará hasta orientarse en el ángulo indicado, medido a partir de la vertical y en el sentido de las agujas del reloj.*




*Efecto sobre la tortuga del comando SETPOS: desplaza a la tortuga en modo absoluto a la posición cuyas coordenadas se incluyen en la lista que acompaña al comando.*

en cualquier momento cuál es el color del fondo. Para ello cuenta con el operador BG (BackGround). BG devuelve el número correspondiente al color actual de la pantalla.

El empleo conjunto del comando SETBG y del operador BG permite múltiples posibilidades. Por ejemplo:

```
SETBG SUM 1 BG
```

recupera el número de color, le suma una unidad, y utiliza el resultado para definir el nuevo color de fondo. Con ello, cada vez que se ejecuta esta orden se altera el color de la pantalla, pasando al color cuyo código coincide con el siguiente en número.

## De lo relativo a lo absoluto

Los desplazamientos producidos por las órdenes FORWARD y BACK se denominan relativos. Este apelativo se refiere al hecho de que la posición final o de destino de la tortuga depende de su situación de partida. Asimismo, la orientación de la tortuga por efecto de LEFT o RIGHT también es relativa: una misma orden LEFT o RIGHT dejará a la tortuga en una orientación distinta según sea la orientación inicial o anterior a recibir la orden. Si lo que se desea es situar a la tortuga en un punto determinado y con una orientación específica, es preciso seguir una serie de pasos:

— Identificar la posición y orientación actuales.
— Orientar a la tortuga en dirección al nuevo punto.
— Avanzar el número necesario de unidades.
— Girar hasta la orientación deseada.

El proceso puede simplificarse mediante el uso previo del comando HOME (traslado al origen), con lo que la posición y orientación inicial de la tortuga resultarán conocidas (las iniciales al encender la máquina). En todo caso, este método supone el paso de la tortuga por el centro de la pantalla.

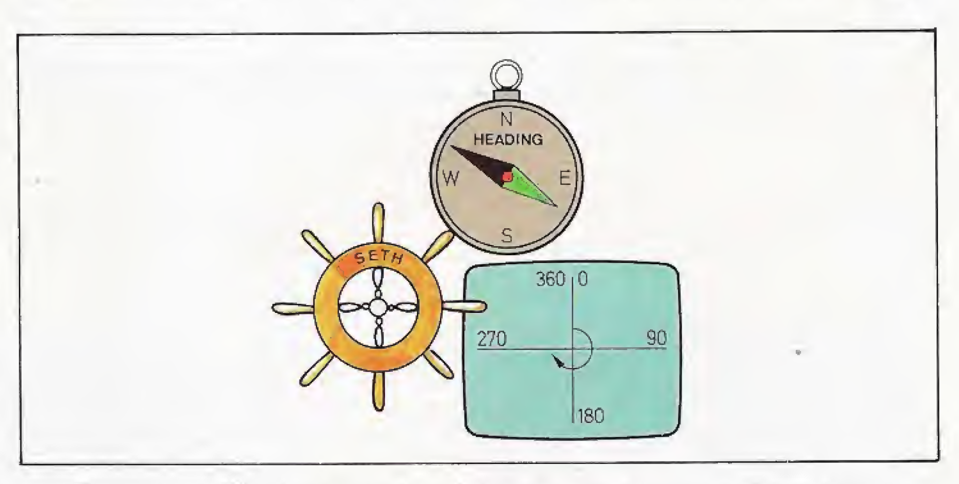
Las órdenes para el desplazamiento absoluto eliminan toda esta suerte de inconvenientes. Con un solo comando, SETPOS, es posible situar a la tortuga en la posición de la pantalla que se desee. Esta debe especificarse por medio de sus coordenadas, expresadas en forma de lista. Dichas coordenadas serán las cartesianas (eje X y eje Y, en ese mismo orden) referidas, por lo general, al centro de la pantalla.

Por ejemplo, la orden SETPOS [50 25] coloca a la tortuga a 50 unidades hacia la derecha y 25 hacia arriba del origen o centro de la pantalla, sea cual fuere su posición actual. Como es habitual, si la tortuga tiene la tiza activada (bajada), aparecerá en la pantalla el trayecto recorrido, que coincidirá con una línea recta. En caso de duda, es posible conocer las coordenadas del punto en el que está situada la tortuga. Al efecto, se dispone del operador POS; éste devuelve una lista conteniendo dichas coordenadas. El uso de POS permite almacenar las coordenadas de un punto de la pantalla para, posteriormente, utilizarlas en orden a ubicar en él a la tortuga por medio de SETPOS. Por ejemplo:

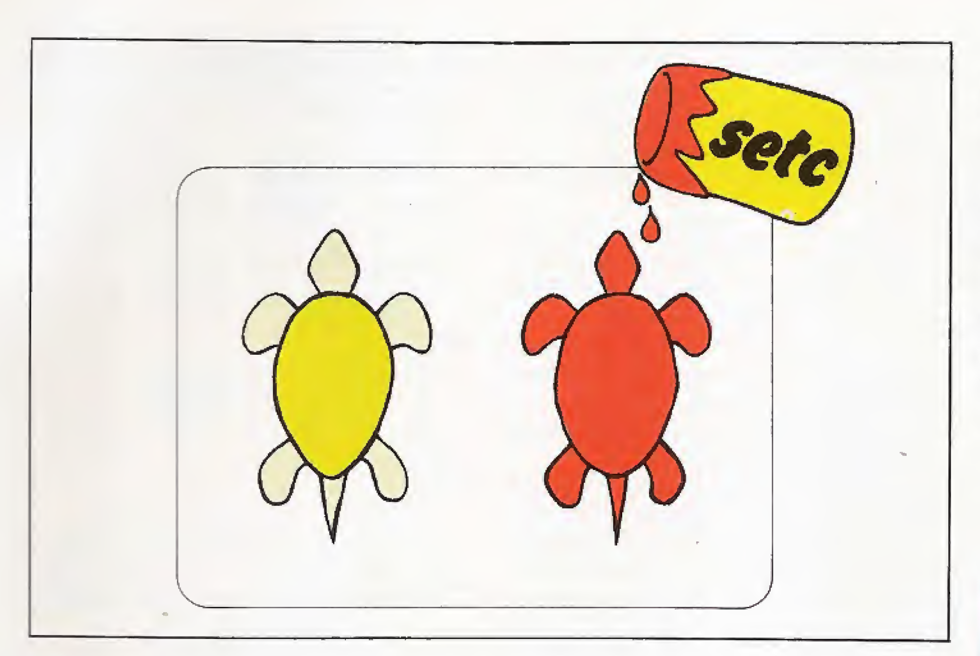
```
PENUP
LEFT 123
FORWARD 73
MAKE "PUNTO POS
HOME
BACK 29
SETPOS :PUNTO
```

En el ejemplo, se selecciona una posición cualquiera por medio de LEFT y FORWARD. Las coordenadas de esta posición se almacenan en la variable PUNTO. A continuación, se utilizan las órdenes HOME y BACK para alejarse del referido punto. Por último, SETPOS :PUNTO restaura a la tortuga en el lugar elegido.

Las dos órdenes mencionadas ejecutan un desplazamiento absoluto definido por las coordenadas. El uso de coordenadas permite definir un nuevo tipo de movimiento: el desplazamiento intermedio entre el absoluto y el relativo. Un desplazamiento de este tipo se obtiene variando únicamente una de las coordenadas. El siguiente procedimiento ilustra tal posibilidad; traslada la tortuga al




*El comando SETH permite orientar a la tortuga en la dirección y sentido definido por el ángulo que constituye su dato de entrada. HEADING es un operador capaz de indicar la orientación actual de la tortuga expresándola en grados.*




*Al igual que el fondo de la pantalla y las tizas de dibujo, también la tortuga puede adoptar diversos colores. El comando al efecto es SETC, seguido por el número correspondiente al color seleccionado.*

punto cuya abscisa (posición X) se indica, sin variar su ordenada (coordenada Y):

```
TO SETPOSX :X
MAKE "Y LAST POS
SETPOS [ :X :Y]
END
```

El cometido del operador LAST es extraer la ordenada de la posición actual; ésta se utiliza al definir la nueva posición, asignando a dicho valor a :Y. un procedimiento complementario a éste será el que altere la coordenada Y, manteniendo la posición horizontal (coordenada X). Respecto al ejemplo precedente, el nuevo procedimiento sustituirá las X por Y, las Y por X y LAST por FIRST.

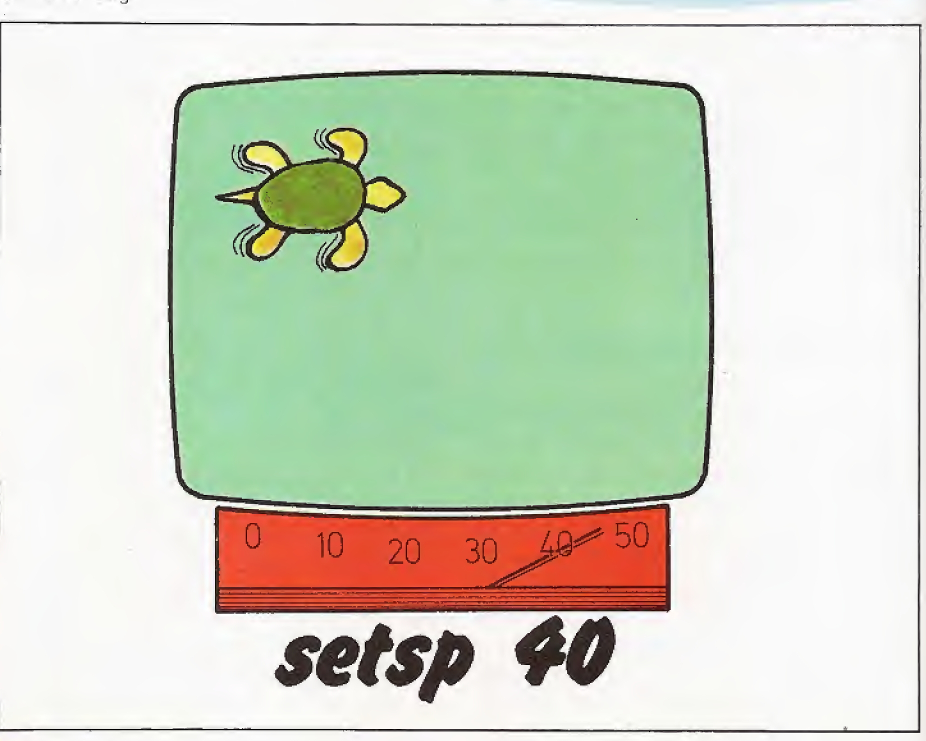
A continuación se introducen nuevas órdenes que permitirán simplificar al máximo este procedimiento. Su cometido es facilitar la reutilización de alguna de las coordenadas de la posición inicial de la tortuga

El lenguaje LOGO incorpora dos operadores útiles al efecto, estos son XCOR e YCOR. El primero devuelve el valor de la abscisa (coordenada X), e YCOR hace lo propio con el valor de la ordenada (coordenada Y). Utilizando esta nueva opción el procedimiento anterior quedará de la siguiente forma:

```
TO SETPOSX :X
MAKE "Y YCOR
SETPOS [ :X :Y]
END
```

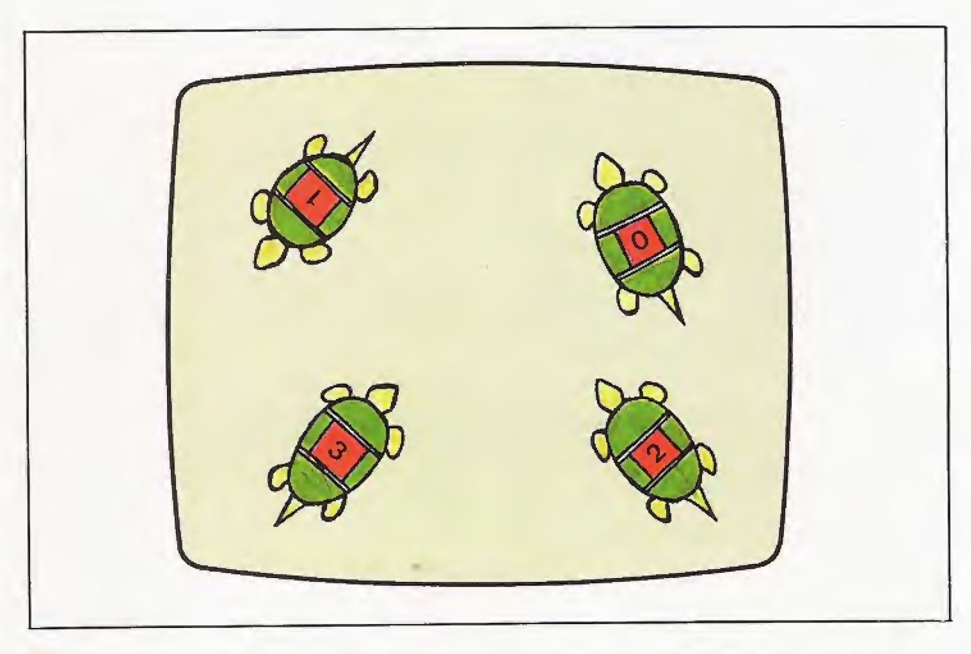
O mejor aún:

```
TO SETPOSX :X
SETPOS [ X :YCOR]
END
```




*El comando SETSP hace que la tortuga se mantenga en movimiento a la velocidad especificada. El operador complementario es SPEED: su ejecución devuelve el valor de la velocidad de desplazamiento.*

Los dos próximos comandos simplifican aún más el procedimiento. En realidad lo reducen a la nada, puesto que realizan específicamente dicho cometido. Estos son SETX y SETY. Con SETX se varía la posición horizontal de la tortuga sin alterar la vertical, mientras que SETY mantiene la posición horizontal (coordenada X) modificando la vertical (coordenada Y).




*El LOGO permite el uso simultánea de varias tortugas. Habitualmente, son cuatro las tortugas utilizables. Para permitir su control, éstas se identifican con un número comprendido entre cero y tres.*

## Cambio de orientación

Las órdenes de movimiento presentadas en los párrafos precedentes, conducen a un cambio de posición que mantiene la *orientación* inicial de la tortuga. Los comandos LEFT y RIGHT, anteriormente comentados, se utilizan para un cambio relativo de orientación: ordenan el giro de la tortuga en un determinado ángulo a partir de la orientación actual.

Para ordenar el posicionamiento en modo absoluto se dispone de SETH (SET Heading). El ángulo de orientación debe ser expresado en grados, tomando como origen (generalmente) el eje vertical, en sentido hacia arriba, y realizándose el giro en el sentido de las agujas del reloj. Así, por ejemplo, SETH 90 coloca a la tortuga mirando hacia la derecha.

Existe, asimismo, un operador que devuelve el ángulo de orientación actual de la tortuga. Este es HEADING; un operador que responde expresando la orientación en los términos antes señalados.

## Control de la velocidad

Una de las características de la tortuga hasta ahora ignorada, es la posibilidad de alterar su velocidad de movimiento. Nuestro disciplinado personaje no tiene por qué permanecer inmóvil, sino que puede lanzarse a recorrer la pantalla con una determinada velocidad. El comando implicado es SETSP (SET SPeed).

Por ejemplo, SETSP 20 hará que la tortuga camine con una velocidad de 20. Desde luego, su paseo no cesará hasta que se le asigne una velocidad cero. Si la tortuga se encuentra con la tiza «bajada», ésta irá dibujando su trayectoria sobre la pantalla. La orden SETSP no altera las restantes condiciones definidas, sino que mantiene los atributos correspondientes a posición, número y color de la tiza. Cabe notar que la velocidad determina un movimiento rectilíneo en la dirección apuntada por el eje longitudinal de la tortuga.

Es necesario significar las distintas reacciones de la tortuga dependiendo del modo en el que estén definidos los límites de la pantalla. Por ejemplo, si se ha seleccionado el modo cerrado (con la orden WRAP), se verá a la tortuga aparecer por el punto opuesto a aquél por donde ha abandonado la pantalla. En el caso de que la tortuga tuviera una orientación oblicua con respecto a los ejes de coordenadas, ésta no correría siempre la misma línea diagonal.

Cuando se opta por la pantalla de límites abiertos (WINDOW), la tortuga desaparece del campo visual y continúa avanzando hasta llegar al final del terreno disponible. La extensión de este terreno depende de la capacidad propia del ordenador. Asimismo, las velocidades máxima y mínima admisibles vienen impuestas por las características del equipo.

Como ya es habitual, el LOGO dispone de la función opuesta. El operador SPEED es el encargado de devolver el indicativo de la velocidad actual. El siguiente procedimiento muestra la acción de las dos órdenes presentadas.

```
TO EJEMPLO
WRAP
MAKE "A [ACELERA]
CORRE
END
```

Como se observa, el procedimiento EJEMPLO incluye en su definición a dos procedimientos elementales, ACELERA y CORRE; a su vez, éste último engloba a un tercer procedimiento elemental: DECELERA. Todos ellos aparecen definidos a continuación.

```
TO CORRE
IF SPEED>100 [MAKE"A[DECELERA]
RUN :A
IF SPEED<1 [STOP]
END
```

```
TO ACELERA
SETSP SPEED+2
END

TO DECELERA
SETSP SPEED-2
END
```

La ejecución del procedimiento EJEMPLO hara que la velocidad de la tortuga aumente de 0 a 100, en pasos de 2 unidades. Al llegar a una velociad mayor que 100, la tortuga empezará a decelerar, hasta que su velociad vuelva a ser cero. Las instrucciones RUN e IF utilizadas en el procedimiento CORRE se describirán en un capítulo posterior dedicado al estudio de los bucles.

## Dibujando con varias tortugas

Una sorpresa agradable la constituye el hecho de que la tortuga pueda también actuar en equipo. El LOGO permite trabajar simultáneamente con más de una tortuga en escena.

Habitualmente, el número máximo de tortugas es de cuatro, identificadas con los números del cero al tres. La tortuga con la que se ha trabajado hasta ahora es la número cero. Para que entren en juego las demás, es preciso «llamarlas» por medio del comando TELL (decir, «dile a»...). Este comando se emplea también para comunicar una orden a una tortuga específica.

Por ejemplo, TELL 1 hace aparecer a la tortuga número 1; a su vez, las órdenes que sigan serán obedecidas por la última tortuga invocada (en este caso, la número 1). Si se pretende que las órdenes afecten simultáneamente a varias tortugas, habrá que advertir de ello a los quelonios. Por ejemplo, para que las tortugas «actuantes» sean las dos primeras, habrá que empezar tecleando TELL [0 1]. En general, es preciso formar una lista con los números de las tortugas que deban obedecer al tiempo las mismas órdenes, y llamarlas con un comando TELL.

Las últimas tortugas invocadas con TELL se denominan *tortugas en curso,* por ser ellas las que obedecen las órdenes posteriores. Si se invoca a varias tortugas situadas en distintas posiciones de la pantalla, lo que se les mande dibujar lo ejecutarán todas ellas a la vez, aunque cada una en la zona que le corresponda. Cambiando de tortuga en curso, es posible controlar a cada una de ellas por separado.

Para identificar a la tortuga o tortugas que están actuando en un determinado momento, hay que hacer uso del operador WHO (quien). Su ejecución responde con el número de la tortuga o tortugas invocadas en último lugar. Este operador resulta útil para dar órdenes dis-

**TABLA DE ORDENES DEL «TURTLE GRAPHICS» (1)**

| Instrucción | Cometido | Operador/ Comando |
|---|---|---|
| HT | Esconde a la tortuga | Comando |
| ST | Muestra a la tortuga | Comando |
| SHOWNP | Devuelve TRUE si la tortuga es visible | Operador |
| WINDOW | Activa el modo en el que la pantalla sólo representa una parte del campo de la tortuga | Comando |
| WRAP | Activa el modo cerrado en el que la tortuga no puede salir de la pantalla | Comando |
| SETPN<NT> | Utiliza la tiza número <NT> | Comando |
| SETPC<NT><NC> | Adjudica a la tiza <NT> el color <NC> | Comando |
| SETBG<NC> | Pone el fondo de color <NC> | Comando |
| PN | Devuelve el número de tiza utilizado | Operador |
| PC<NT> | Devuelve el número del color de la tiza <NT> | Operador |
| BG | Devuelve el número del color de fondo | Operador |
| <NT>: número de tiza.<br><NC>: número de color. | | |

**TABLA DE ORDENES DEL «TURTLE GRAPHICS» (2)**

| Instrucción | Cometido | Operador/ Comando |
|---|---|---|
| SETPOS<lista> | Sitúa a la tortuga en la posición dada por las coordenadas de la lista | Comando |
| POS | Devuelve una lista conteniendo las coordenadas del punto donde se encuentra la tortuga | Operador |
| XCOR | Devuelve el valor de la coordenada X del punto actual | Operador |
| YCOR | Devuelve el valor de la coordenada Y del punto actual | Operador |
| SETX<número> | Sitúa a la tortuga en el punto de coordenada X, dada por <número>, sin variar su coordenada Y | Comando |
| SETY<número> | Sitúa a la tortuga en el punto de coordenada Y, dada por <número>, sin variar su coordenada X | Comando |
| SETH<número> | Sitúa a la tortuga con la orientación dada por <número> | Comando |
| HEADING | Devuelve la orientación actual de la tortuga | Operador |

tintas a cada tortuga. Por ejemplo, una instrucción del tipo FORWARD WHO*10, hará que si la tortuga en curso es la uno, ésta avance 10 posiciones, si es la dos, 20...

## Tortugas bajo control

Para dirigir a cada tortuga por separado se hace necesario llamarlas alternativamente. Ello se puede conseguir con el uso reiterado del comando TELL, o escribiendo procedimientos que utilicen dicho comando.

El siguiente ejemplo cambia la tortuga actuante por la identificada por el siguiente número.

```
TO CAMBIATOR
IF WHO=3[TELL 0] [TELL WHO+1]
END
```

Cada vez que se ejecute el procedimiento cambiará la tortuga en curso. Con éste y otros procedimientos semejantes, el programador estará en condiciones de alterar la situación de las tortugas, para que cada una realice una tarea distinta. Existen también otros comandos que permiten una mayor flexibilidad en el uso de varias tortugas. Estos son los que se detallan a continuación.

Cuando las órdenes a ejecutar por cada tortuga sean idénticas, se puede utilizar EACH (cada). La sintaxis de este comando es la siguiente: EACH <lista de instrucciones>. Por ejemplo, EACH [FORWARD 200] hará que *cada* tortuga avance 200 posiciones. La diferencia con la situación normal consiste en que, ahora, la segunda tortuga no cursa la orden hasta que termina de hacerlo la primera. De no utilizar el comando EACH, todas las tortugas «acatarían» la orden a la vez.

La principal utilidad de EACH radica en su empleo conjunto con WHO. Tal

**TABLA DE ORDENES DEL «TURTLE GRAPHICS» (3)**

| Instrucción | Cometido | Operador/ Comando |
|---|---|---|
| SETSP<número> | Hace que la tortuga se mantenga en movimiento a la velocidad dada por <número> | Comando |
| SPEED | Devuelve la velocidad actual de la tortuga | Operador |
| TELL<nu/lis> | Define qué tortugas serán las actuantes a partir de ese momento | Comando |
| WHO | Devuelve una lista con los números de las tortugas en curso | Operador |
| EACH...<lista ins> | Hace que las tortugas en curso ejecuten la lista de instrucciones, una tortuga después de otra | Comando |
| ASK<nu/lis> <lista ins> | Hace que la o las tortugas especificadas ejecuten las instrucciones de la lista | Ambos* |
| SETC<número> | Pone a la tortuga en curso del color indicado por <número> | Comando |
| COLOR | Devuelve el número que indica el color de la tortuga en curso | Operador |

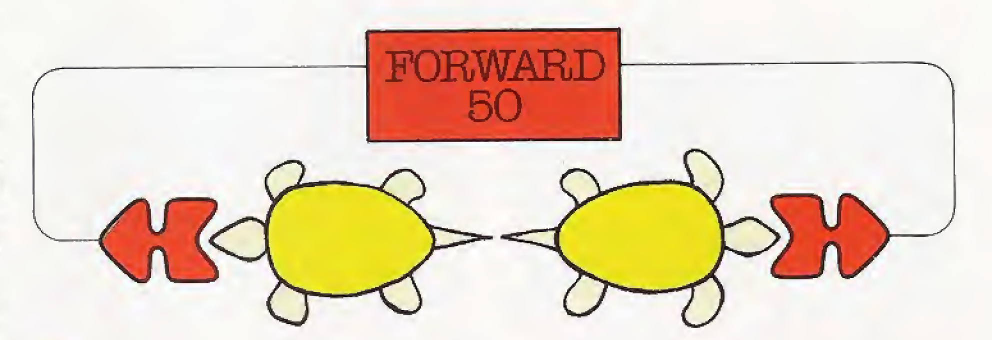
<nu/lis>: número de tortuga o lista de números de tortugas.
<lista ins>: lista de instrucciones.
* ASK se comporta como operador o comando dependiendo de la lista de instrucciones que lo acompañen.

combinación permitirá ejecutar órdenes distintas a cada tortuga; por supuesto, siempre que las órdenes se puedan reducir a una función del número de cada tortuga.

Por ejemplo, para que cada tortuga gire en un ángulo de distinto número de grados, se puede introducir la orden: EACH [RIGHT WHO*90]. Con el ejemplo apuntado más arriba, en el que se utilizaba WHO sin EACH, se producirá un error si hay más de una tortuga en acción, simultáneamente. Ello se debe a que WHO devuelve una lista de números. Sin embargo, cuando actúa en compañía de EACH, el LOGO se encarga de extraer de la lista cada número por separado.

EACH no cambia la tortuga en curso y sólo actúa con las que estén disponibles en ese instante.

El método adecuado para emplear una tortuga que no sea la actualmente en curso, sin por ello cambiarla, llega de la mano de ASK. Este comando exige dos datos de entrada: el primero es el número de tortuga y el segundo la lista de las instrucciones que ha de «obedecer». Si se desea que las instrucciones las ejecuten varias tortugas, el primer dato debe ser una lista con los números de las tortugas afectadas.

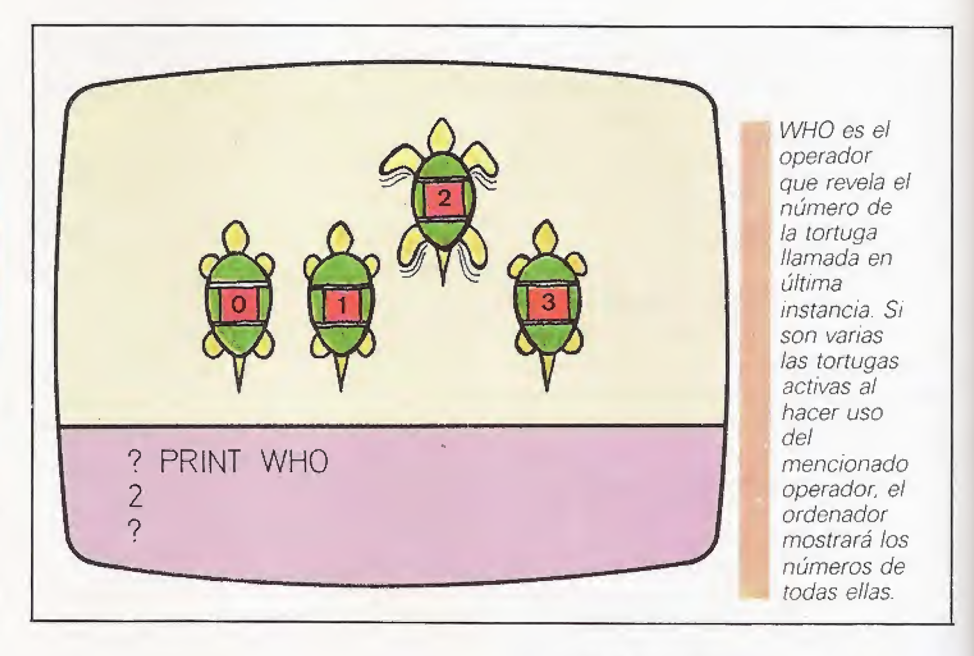



*La misión del comando TELL es definir qué tortugas van a actuar a partir de ese instante. Tras llamar a dos tortugas —localizadas en distintos puntos de la pantalla o con distinta orientación— por medio del mencionado comando y ordenar un determinado movimiento, éstas ejecutarán el desplazamiento por distintas zonas de la pantalla, de acuerdo con su orientación y posición de partida.*

El siguiente ejemplo ilustra la actuación de cada comando.

```
TELL [0 1]
PRINT WHO
0 1

EACH [SETX WHO*50]
ASK 2 [BACK 50]
PRINT WHO
0 1
```




*WHO es el operador que revela el número de la tortuga llamada en última instancia. Si son varias las tortugas activas al hacer uso del mencionado operador, el ordenador mostrará los números de todas ellas.*

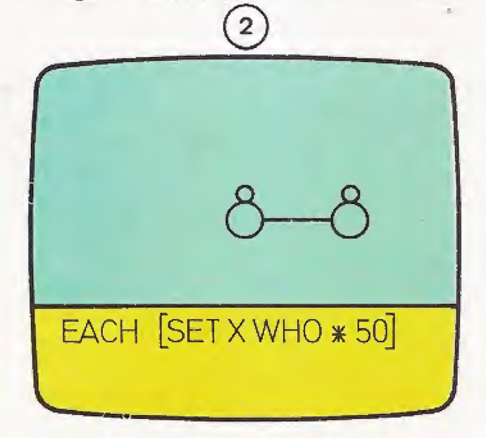
Como se observa, ASK no altera las tortugas en curso.

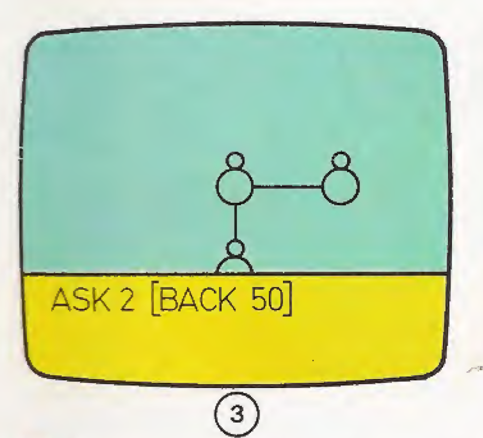
Otra posibilidad de ASK radica en su empleo como operador. Realmente, la acción de ASK es la que defina la lista de instrucciones que constituyen su dato de entrada. Ello significa que se comportará como un operador si la lista da un dato de salida. He aquí un ejemplo:

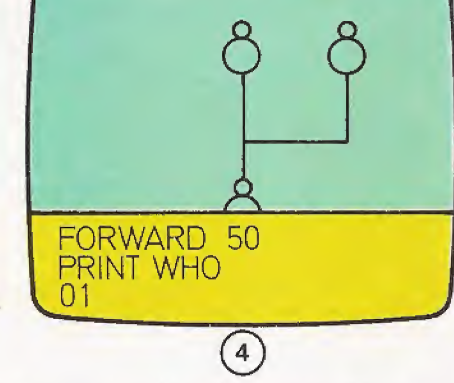
```
PRINT ASK 1 [POS].
```

En este caso, las coordenadas de la tortuga 1 constituyen el dato de salida, a raíz de lo cual, ASK actúa como operador.




*Secuencia de pantallas que ilustran el efecto de las órdenes TELL, ASK y EACH.*

## Tortugas de colores

Al igual que el fondo y las tizas, también las tortugas pueden cambiar de color.

Las instrucciones al efecto son SETC y COLOR. SETC, seguido por un número, otorgará a la tortuga o tortugas en curso el color especificado por dicho número.

A su vez, el operador COLOR devuelve el número o código del color adoptado por las tortugas actualmente en uso.

Ambas órdenes pueden asociarse tanto con EACH como con ASK. Una observación a considerar es que el color de las tortugas es totalmente independiente de los colores elegidos para el fondo y las tizas.

## ¿Una o varias?

Todas las instrucciones presentadas para su ejecución con una sola tortuga pueden ser utilizadas con varias. Al respecto, sólo habrá que prever el uso de TELL, con lo que todas las tortugas en curso obedecerán las órdenes del programa. Cabe señalar también que por medio de ASK y EACH será posible diferenciar distintas órdenes para cada tortuga.

# LOGO (4)

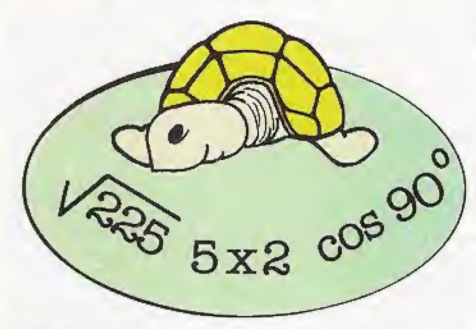
## Aritmética y estructuras de control



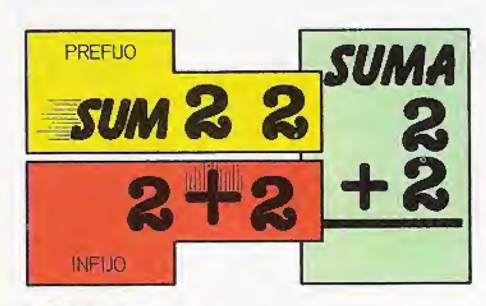
Hasta ahora sólo se han analizado a fondo las capacidades gráficas y de tratamiento de palabras del LOGO. Aunque ambos aspectos son primordiales en este lenguaje, queda aún por comentar otra vertiente fundamental: el tratamiento de datos aritméticos y lógicos. El ordenador es, básicamente, una máquina capacitada para realizar cálculos; en consecuencia, todo lenguaje de programación debe incluir un amplio repertorio de instrucciones aritméticas.

El cálculo numérico no es una posibilidad fácil de explotar en el LOGO. Este lenguaje, al contrario de otros como el FORTRAN o el propio BASIC, no está especialmente orientado al cálculo. No obstante, aporta suficientes funciones aritméticas para realizar la mayor parte de las operaciones habituales.

En otros capítulos se ha hecho referencia a los bucles, y será en éste donde se tratarán a fondo junto con mecanismos avanzados de decisión.




*Aunque no fue específicamente creado para el cálculo matemático, el lenguaje LOGO permite programar operaciones aritméticas con comodidad.*




*Muchos de los operadores matemáticos del LOGO pueden adoptar una doble formulación: como «prefijo», precediendo a los datos a operar, o «infijo», colocado entre los operadores.*

## Prefijos e infijos

Las órdenes LOGO pueden ser comandos u operadores. Las funciones aritméticas se encuadran en este segundo grupo. El primer operador aritmético a analizar es el de suma, materializado por medio de la palabra SUM. Como todo operador LOGO tiene sus entradas y salidas. Las entradas de SUM se colocan a la derecha de la orden: SUM 2 3 es un posible ejemplo. Al igual que cualquier otro operador, debe estar precedido por un comando que «recoja» su dato de salida.

Dicha forma de realizar la suma y devolver el resultado se denomina *modo prefijo,* debido a que el operador *precede* a los comandos. Este es el método habitual de utilizar los operadores en LOGO; si bien, puede resultar engorroso en el caso de los operadores aritméticos. Esta es la razón de que exista otra posibilidad, el *modo infijo.* La nueva modalidad consiste en colocar el operador *entre* los operandos. En tal caso, hay que sustituir la palabra SUM por el signo +, por ejemplo: ...2 + 3. Su efecto es idéntico al del ejemplo anterior, aunque su significado es más familiar para el usuario.

Entre los operadores que permiten ambos modos de formulación, se encuentran: SUM (+), PRODUCT (*) y EQUALP (=). Junto a los *prefijos* se indican, entre paréntesis, los correspondientes *infijos.*

## Aritmética

Las operaciones básicas del LOGO son las siguientes: suma (+), resta (–), multiplicación (*) y división (/). Todas ellas disponen de un operador infijo (entre paréntesis). Sólo el producto (PRODUCT) y la suma (SUM) admiten prefijo. Hay que señalar que algunos dialectos del LOGO incluyen un prefijo para división, coincidente con la palabra QUOTIENT.

Los operadores SUM y PRODUCT ofrecen, además, la posibilidad de aceptar más de dos datos de entrada. Ello se consigue encerrando el operador y sus datos de entrada entre paréntesis. Por ejemplo: (PRODUCT 2 3 4), cuyo dato de salida será 24 (2 × 3 × 4).

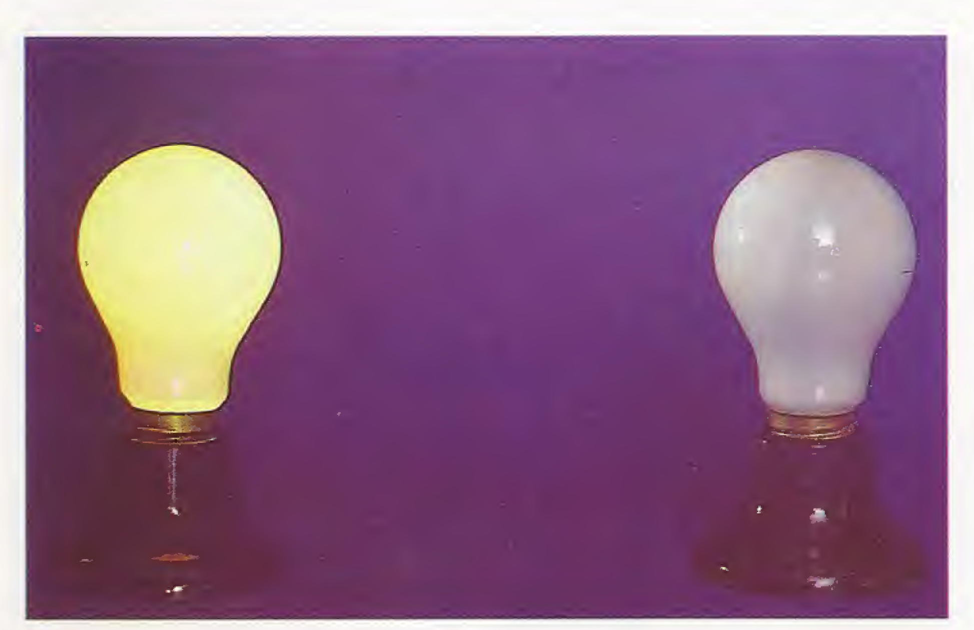
Además de las operaciones elementales, el LOGO incluye algunas funciones matemáticas complejas. Estas son la raíz cuadrada (SQRT) y las funciones trigonométricas seno y coseno (SIN, COS). Todas ellas admiten un solo dato de entrada y, por lo tanto, deben ser formuladas como prefijos. Otra función útil en múltiples casos en REMAINDER. Este operador calcula el resto ocasionado por la división de los datos colocados a su derecha. Es, por lo tanto, un operador en modo prefijo.

Dentro de los operadores aritméticos se pueden considerar los de «truncamiento». Es decir, aquellos que extraen una parte del número indicado. Estos son INT y ROUND. El primero selecciona la parte entera del dato de entrada, mientras que ROUND devuelve el número entero «más cercano» al dato de entrada. Para el dato 15.7, INT devuelve el valor 15; ROUND, por el contrario, daría como resultado el valor 16.

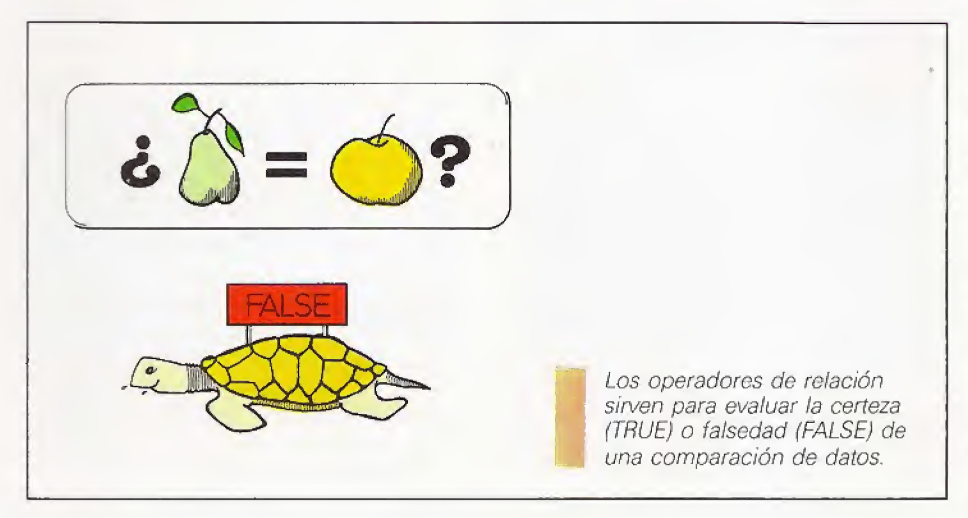
Todos estos operadores son utilizados, generalmente, para la asignación de valores a una variable. he aquí algunos ejemplos.

```
MAKE "A 2+3
MAKE "SUMA SUM :SUMA :DATO
MAKE "COSA COS :A
MAKE "RESUL INT (PRODUCT 2 :A SIN :A
MAKE "TON 2+:PE/(5-:A)
```

Como puede apreciarse en los ejemplos, es posible encadenar operadores. Cuando los operadores encadenados sean infijos, se evaluarán primero los productos y divisiones y, más tarde, las sumas y restas. En la última línea del ejemplo, se utilizan paréntesis para que el contenido de la variable :PE sea dividido por el resultado de 5 – :A. Si no se coloca el paréntesis, se restaría :A del cociente de :PE/5.



*Las comparaciones de datos en LOGO conducen a dos valores de salida de tipo «lógico»: TRUE (cierto) y FALSE (falso).*




*Los operadores de relación sirven para evaluar la certeza (TRUE) o falsedad (FALSE) de una comparación de datos.*



*El operador RANDOM genera un número aleatorio, comprendido entre cero y el valor especificado como dato de entrada.*

## Numeros aleatorios

El lenguaje LOGO dispone también de un generador de números aleatorios, muy útil en juegos y simulaciones de azar. El operador RANDOM, seguido por un número entero, entregaría un resultado aleatorio (también número entero) comprendido entre 0 y el dato de entrada. Por ejemplo, RANDOM 5 dará como dato de salida un entero menor que 5.

```
REPEAT 4 (PRINT RANDOM 5)
2
4
3
2
```

Una determinada secuencia de números aleatorios puede ser repetida por medio de RERANDOM. La instrucción de este comando hace que el ordenador «recuerde» la siguiente secuencia aleatoria generada; cada vez que se vuelve a ejecutar el comando RERANDOM, los números aleatorios generados serán repetidos en el orden en que lo fueron anteriormente. Para la repetición de una secuencia alaeatoria, es imprescindible que el argumento de RAMDOM coincida con el propuesto en la sentencia recordada. El ejemplo ilustra esta posibilidad.

```
RERANDOM REPEAT 4 (PRINT RANDOM 5)
1
3
4
2

RERANDOM REPEAT 4 (PRINT RANDOM 5)
1
3
4
2
```

## Aritmética y listas

La potencia del LOGO en el tratamiento de listas puede ser aprovechada

para fines matemáticos. Esta situación nace de la posibilidad de tratar listas de números. A una lista formada por números pueden aplicársele todos los operadores mencionados en el capítulo de palabras y listas. Por ejemplo, FIRST (1 2 3 4) devolverá el número uno. Este dato de salida puede ser posteriormente procesado por operadores aritméticos. El siguiente procedimiento suma los elementos de una lista de números.

```
TO SUMA: LISTA
IF: EMPTYP: LISTA(OUTPUT O) (PRINT SUM FIRST:LISTA
SUMA BUTFIRST:LISTA)
END
```

El significado completo del procedimiento se comprenderá al tratar la recursividad en el apartado de bucles. Sirva de todas formas este ejemplo para ilustrar la posibilidad de operar con listas de números.

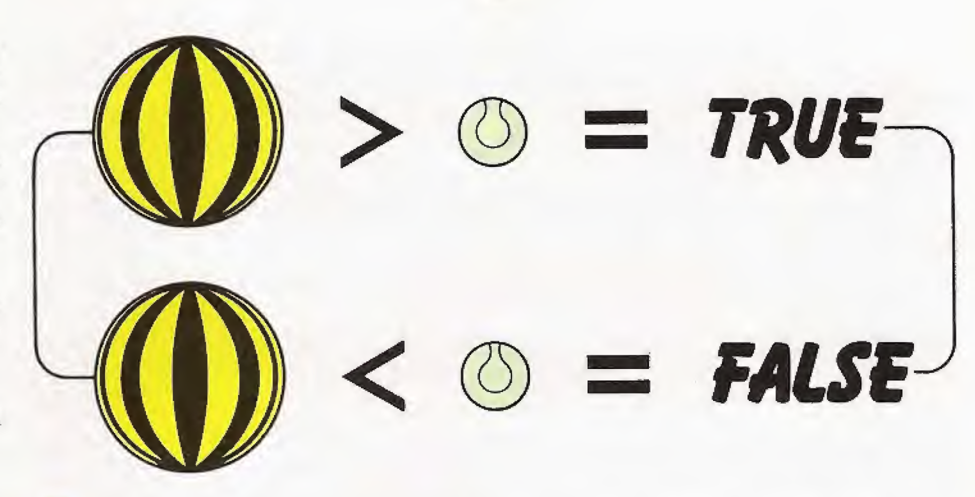
## Comparaciones

El mundo está lleno de comparaciones, derivadas de las diferencias entre los diferentes individuos. Una persona puede ser más alta que otra, o más obesa. En el terreno matemático estas comparaciones son un calco de las del mundo real. En muchos problemas matemáticos resulta imprescindible determinar si un número x es mayor que otro y. El LOGO aporta tres operadores de relación para la comparación de números: mayor que (>), menor que (<) e igual (=). Los tres actúan como infijos, aunque el tercero admite también la modalidad prefijo; en tal caso su sintaxis es EQUALP. El dato de salida de estos operadores recibe el nombre de dato *lógico,* puede adoptar el valor TRUE (verdadero) o FALSE (falso). En esta tesitura, la salida de 5 > 6 será FALSE mientras que la de EQUALP 5 5 será TRUE.

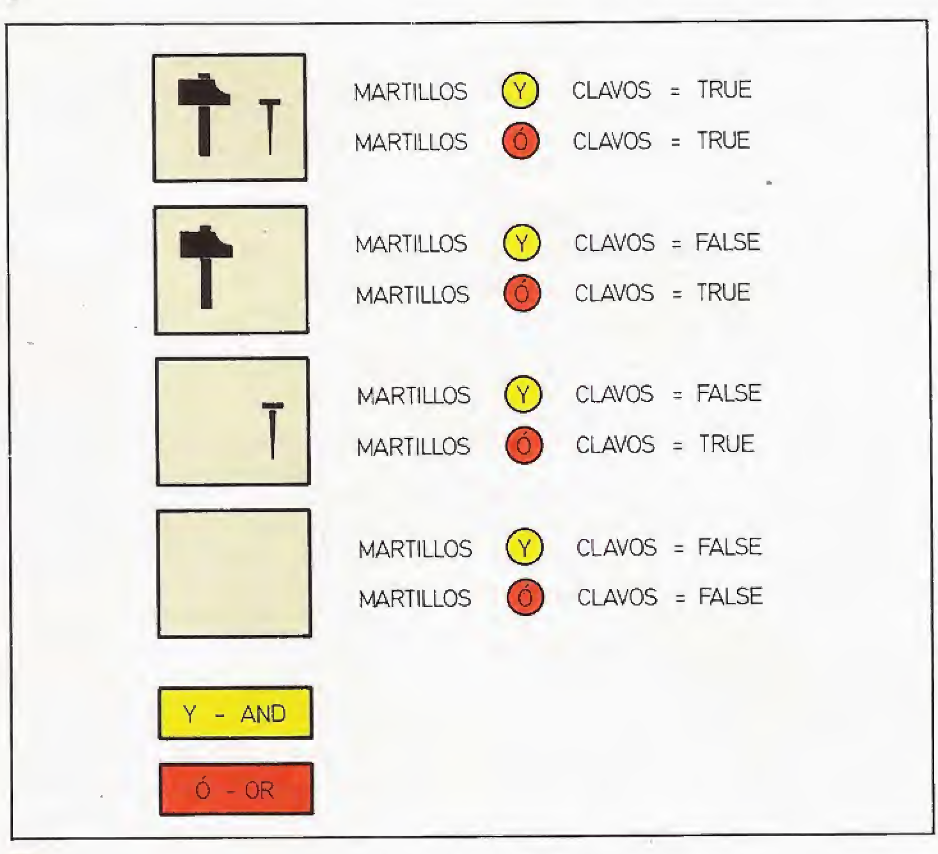
Otros operadores de este tipo —con dato de salida de tipo lógico— han sido estudiados en capitulares anteriores: LISTP y WORDP al hablar de palabras y listas; NAMEP, al hablar de variables: SHOWNP en los TURTLE GRAPHICS...

Los datos de salida de tipo lógico son palabras manejables por el LOGO. De hecho, si se precede un operador de este tipo por un comando PRINT, en la pantalla aparecerá la palabra lógica correspondiente. Por ejemplo: PRINT 5 > 6 escribe en la pantalla la palabra FALSE.

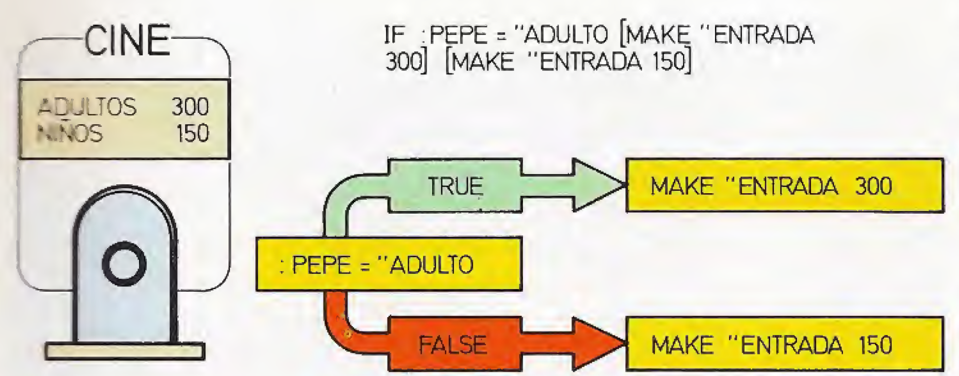
También se pueden utilizar estos datos de salida para formar listas, e incluso procesarlas tal y como se explicó en el capítulo dedicado a palabras y listas. Los datos lógicos tienen además otro significado para el intérprete LOGO. Esta segunda característica de las palabras TRUE y FALSE es la que se detalla en el siguiente apartado.



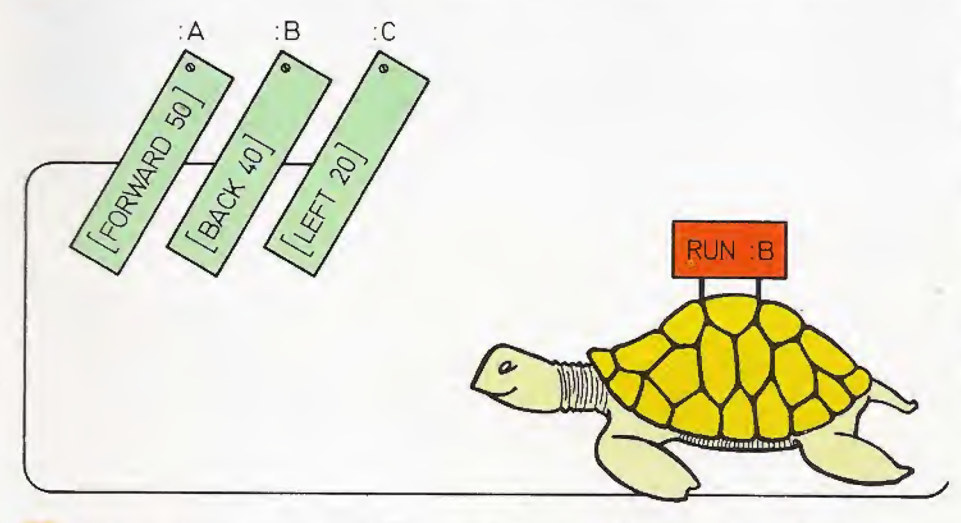

*Funcionamiento de los operadores de relación «mayor que» y «menor que».*




*Salida generada por los operadores lógicos AND y OR ante diversas combinaciones de datos de entrada.*




*IF es una orden LOGO adecuada para bifurcar el flujo normal de control de un programa, de acuerdo con el cumplimiento de una condición. En el ejemplo, la condición ADULTO determina qué dato se asignará a la variable "ENTRADA".*




*RUN es una nueva orden LOGO especializada en la ejecución de una lista de instrucciones. Esta lista, incluida como dato de entrada de RUN, puede verse reemplazada por la correspondiente variable representativa de la lista a ejecutar.*

## Operadores lógicos

Una de las teorías matemáticas más utilizadas en el mundo de los ordenadores es la denominada álgebra de proposiciones o *álgebra de Boole*. Esta parte de la matemática moderna se ocupa de las relaciones entre los enunciados lógicos. En ella existen dos operadores lógicos fundamentales AND (y) y OR (o). El cometido de ambos salta a la vista al evaluar la diferencia entre la frase «comeré carne y patatas» y la frase «comeré carne o patatas».

Al tratar con datos de tipo lógico es cuando estos operadores cobran importancia. La instrucción AND :B=0 :C=0 dará el valor TRUE cuando *ambas* variables B y C valgan cero. Realmente, los datos de entrada al operador AND son datos de tipo lógico; en el caso del ejemplo, coinciden con los datos de salida de dos comparaciones.

Si tanto B como C valen cero, el ejemplo anterior equivale a la instrucción AND TRUE TRUE. Por el contrario, si C tuviese un valor distinto de cero, el equivalente sería AND TRUE FALSE.

Estas mismas normas rigen para el uso de OR. La diferencia radica en que, mientras AND exige que sus dos entradas sean TRUE para dar una salida TRUE, a OR le basta con que una de ellas sea TRUE. En el ejemplo comentando al principio, la primera frase será cierta si el individuo come ambas cosas, carne y patatas, y falsa si deja de comer alguna de ellas. Por el contrario, la segunda frase será cierta siempre que coma alguna de las viandas mencionadas y falsa sólo en el caso de que no se coma ninguna de ellas.

Las tablas adjuntas muestran la salida que proporciona cada uno de estos operadores para el conjunto de posibles combinaciones de entrada.

Como quiera que su salida es otro dato lógico, resulta evidente que estos operadores puedan ser encadenados, obteniendo así funciones más complejas. Por ejemplo, la posibilidad de comer «sopa y carne o pescado, o arroz y carne con patatas» se formularía de la siguiente forma: OR (AND :SOPA OR :CARNE:PESCADO) (AND :ARROZ AND :CARNE:PATATAS).

La certidumbre de dichos enunciados dependerá del valor lógico de las variables :SOPA, :CARNE, etc.

Existe un tercer operador lógico. Este actúa sobre un dato lógico, negándolo, esto es, invirtiendo su valor lógico. El mencionado operador se denomina NOT (no) y su tabla correspondiente es la que aparece en la figura.

Como se puede apreciar en la tabla, el operador NOT admite un único dato de entrada, entregando el valor lógico contrario.

Estos tres operadores son suficientes para relacionar todo tipo de enunciados.

## Bucles

Un hecho evidente es que el ordenador ha sido creado para descargar al usuario de las tareas rutinarias. Este concepto es ampliamente contemplado en la programación de bucles.

En el siguiente ejemplo se utiliza a la tortuga para dibujar un cuadrado:

```
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
```

Se observa que las instrucciones FORWARD 50 y RIGHT 90 se repiten en cuatro ocasiones. El usuario se ve obligado a teclear cuatro veces las mismas órdenes.

Este hecho queda subsanado con el uso de REPEAT.

El comando REPEAT es la primera orden especializada en la creación de bucles. Su presencia simplifica el ejemplo anterior a su mínima expresión:

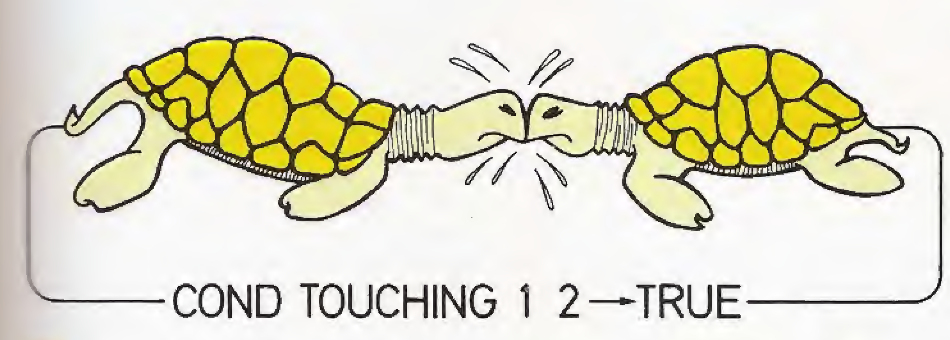
```
REPEAT 4 [FORWARD 50 RIGHT 90]
```

En efecto, una sola línea de instrucción suple a las ocho que constituían la primera alternativa para dibujar el cuadrado. El comando REPEAT admite dos datos de entrada. El primero ha de ser un dato numérico que especifica el número de veces que debe repetirse una acción. Dicha acción se introduce como segundo dato de entrada. Este último puede coincidir con una lista de instrucciones.

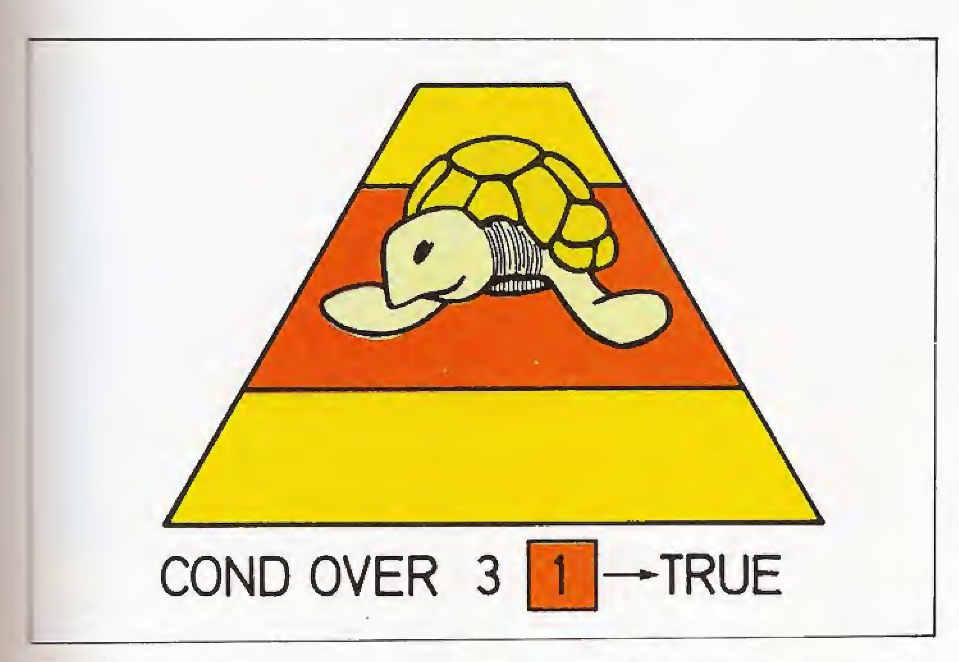
El procedimiento que sigue, POLI, hace uso de REPEAT para dibujar un polígono:

```
TO POLI :LONG :NUM
REPEAT :NUM [FORWARD :LONG
RIGHT (360/:NUM)]
END
```

El dato LONG define la longitud del lado, mientras que NUM indica el número de lados del polígono. El cuadrado anterior puede también confeccionarse por medio de POLI 50 4.




*La propia tortuga del LOGO puede introducir criterios para bifurcar la ejecución: por ejemplo, a través de la orden COND, es capaz de detectar la ocurrencia o no del tipo de colisión definida.*




*OVER es un operador LOGO cuyo cometido es devolver el código de colisión entre la tortuga número T (3 en el ejemplo), y un trazo creado por medio de un determinado número de tiza (tiza número 1 en la figura).*

## Bifurcaciones

En algún caso, puede ser necesario que dentro de un procedimiento se realicen distintas acciones, dependiendo de ciertas condiciones impuestas. Por ejemplo, se puede definir un procedimiento que calcule raíces cuadradas mediante el operador SQRT:

```
TO RAIZ :N
PRINT SQRT :N
END
```

Este procedimiento dará error si el número introducido es negativo (no es posible hallar la raíz cuadrada de un número negativo). Para evitarlo, puede ordenarse que el cálculo de la raíz cuadrada se efectúe exclusivamente si el número N es positivo.

```
TO RAIZ :N
IF :N>0 [PRINT SQRT :N]
END
```

En este segundo caso sólo se calcula la raíz si el número es mayor que cero (N>0). La orden IF sirve para «bifurcar» la ejecución. Esta orden admite tres datos de entrada. El primero corresponde a una condición, esto es: a un enunciado lógico cuya evaluación dará como resultado un dato de tipo lógico. Dependiendo de si el valor resultante es TRUE o FALSE, se ejecutará una lista de instrucciones u otra. Ambas listas de instrucciones constituyen los restantes datos de entrada de la orden IF. Siguiendo con el mismo ejemplo, veamos cuál es el nuevo procedimiento, ligeramente más complejo:

```
TO RAIZ :N
IF :N>0 [PRINT SQRT :N] [PRINT
[SOLO NUMEROS POSITIVOS]]
END
```

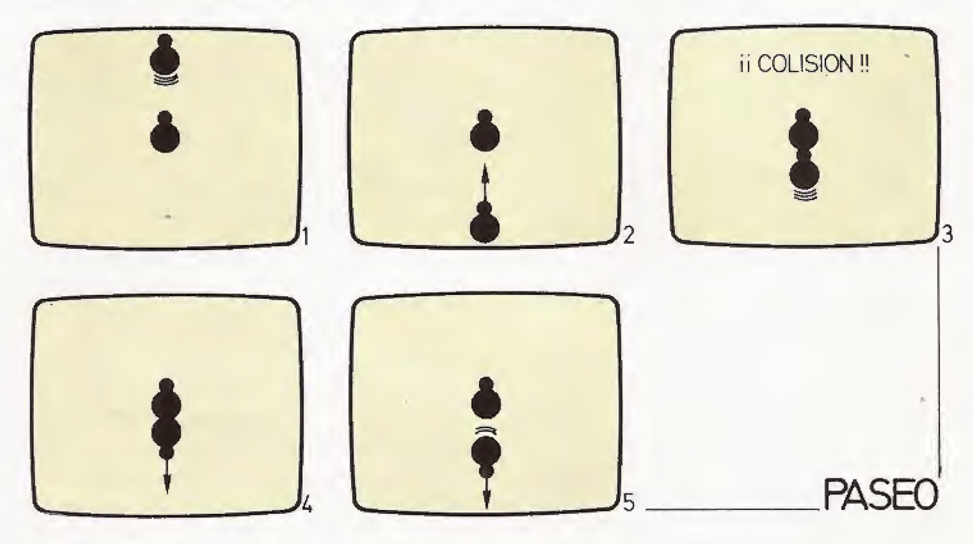
En el primer caso sólo se utilizó una lista de instrucciones, mientras que ahora se contemplan ambas posibilidades. Si :N no es mayor que 0, se ejecutará la segunda lista de instrucciones.

El enunciado de la condición ha de producir un dato de tipo lógico. En consecuencia, este enunciado puede incluir todo tipo de operadores lógicos. Aquí es donde los operadores AND, OR, NOT adquieren una importancia especial. Suponga, por ejemplo, que es preciso que N sea mayor o igual que 0. Tal condición queda plasmada en la expresión siguiente:

```
IF OR :N  0 :N =0...
```

De igual forma, también está permitido el uso de otros operadores como LISTP, WORDP, SHOWNP, etc., para bifurcar la ejecución. Estos últimos permiten el tratamiento diferenciado de los distintos tipos de datos.

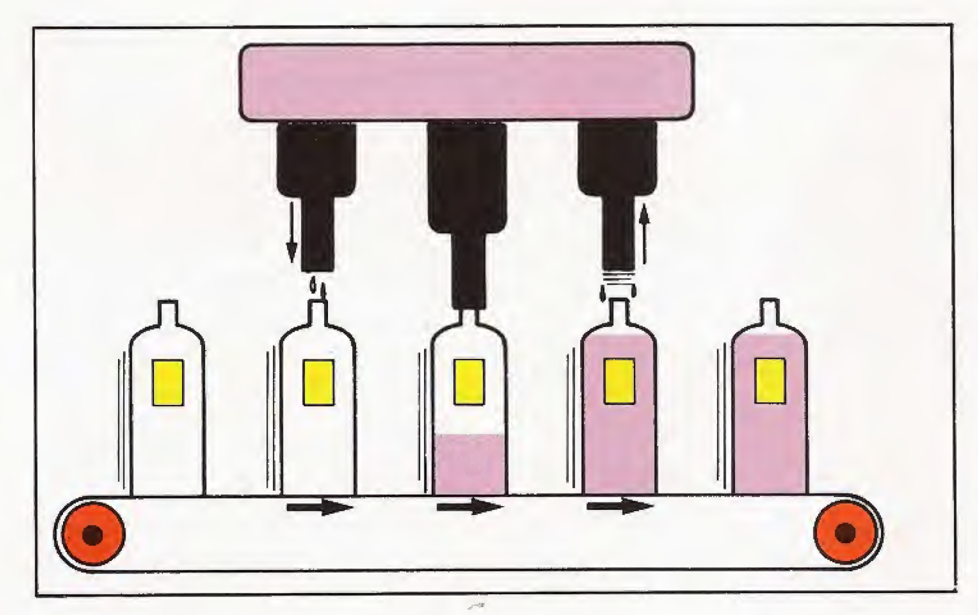
```
TO RAIZ :N
IF NOT NUMBERP :N [PRINT [HAY QUE APORTAR UN NU
MERO]] [IF OR :N>0 :N =0
[PRINT SQRT :N] [PRINT [SOLO NUMEROS POSITIVOS]]]
END
```




Secuencia de ejecución del procedimiento PASEO.

Ahora puede verse la utilidad de operadores como NUMBERP. La ejecución de este último procedimiento aparece en la siguiente pantalla.

```
RAIZ "CUATRO
HAY QUE APORTAR UN NUMERO
RAIZ–4
SOLO NUMEROS POSITIVOS
RAIZ 4
2
RAIZ 0
0
```



*La programación de tareas repetitivas se realiza recurriendo a los bucles o estructuras cíclicas. El lenguaje LOGO también dispone de órdenes adecuadas para la puesta en práctica de este tipo de procesos.*

La instrucción IF actuará como comando u operador según la naturaleza de las órdenes contenidas en las listas. En los ejemplos precedentes realizaba una acción propia de un comando, desencadenada por la presencia de PRINT. Veamos un nuevo ejemplo en el que actúa a modo de comando:

```
TO RAIZOP :N
IF NOT NUMBERP :N [OUTPUT [HA DE APORTAR UN NU
MERO]] [IF OR :N>0 :N =0
[OUTPUT SQRT :N][OUTPUT [SOLO NUMEROS POSITIVOS]]]
END
```

Su ejecución será:

```
PRINT RAIZOP "CUATRO
HA DE DARSE UN NUMERO
PRINT RAIZOP–4
SOLO NUMEROS POSITIVOS
RAIZOP 4
YOU DON'T SAY WHAT TO
DO WITH 2
PRINT RAIZOP 4
2
PRINT RAIZOP 0
0
```

Al actuar en modo operador, IF entrega un dato de salida. Este dato debe constituir la entrada a un comando; en caso contrario, el LOGO escribirá en la pantalla un mensaje de error.

## Ejecución de una lista

Tal como se ha estudiado, las órdenes REPEAT e IF admiten una lista de órdenes como dato de entrada. En ambos casos, la lista incluye las instrucciones a ejecutar.

Cabe observar que el contenido de la lista no tiene por qué ser fijo. Es posible hacer uso de una variable cuyo contenido coincida con la lista específica de órdenes. El siguiente procedimiento ilustra lo indicado:

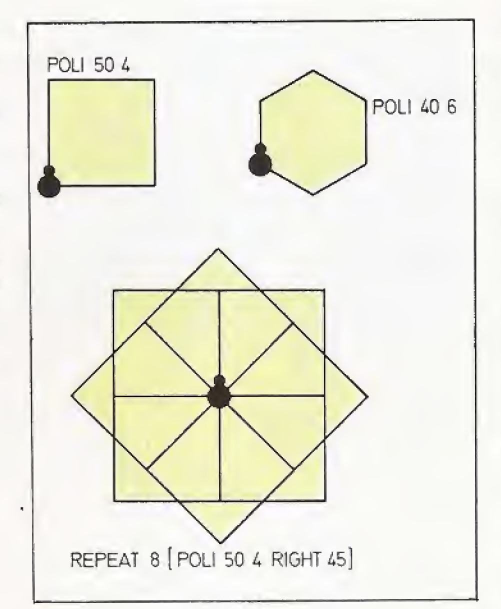
```
TO NUMBER :N
MAKE "POSI [PRINT "POSITIVO]
MAKE "NEGA [PRINT "NEGATIVO]
IF :N<0 :NEGA :POSI
END
```

El procedimiento utiliza dos variables (:NEGA y :POSI) para almacenar la lista de instrucciones.

Existe otro comando LOGO adecuado para ejecutar una lista de instrucciones: RUN.

Su dato de entrada debe coincidir con la lista a ejecutar. Normalmente, la lista se verá reemplazada por una variable que adecuará su contenido para que se ejecute una y otra lista de órdenes LOGO.

```
TO NUMBER :N
MAKE "POSI [PRINT "POSITIVO]
MAKE "NEGA [PRINT "NEGATIVO]
IF :N<0[MAKE "L :NEGA]
[MAKE "L :POSI]
RUN :L
END
```




*El procedimiento POLI, definido en el texto, incluye un bucle construido con la colaboración de la estructura REPEAT. La figura muestra algunos ejemplos prácticos basados en dicho procedimiento.*

## Colisiones

La tortuga también puede introducir criterios para bifurcar la ejecución. Al margen de las funciones lógicas ya estudiadas, existe una nueva función adecuada para detectar colisiones. El operador COND devuelve un valor lógico indicando si se produce la colisión definida.

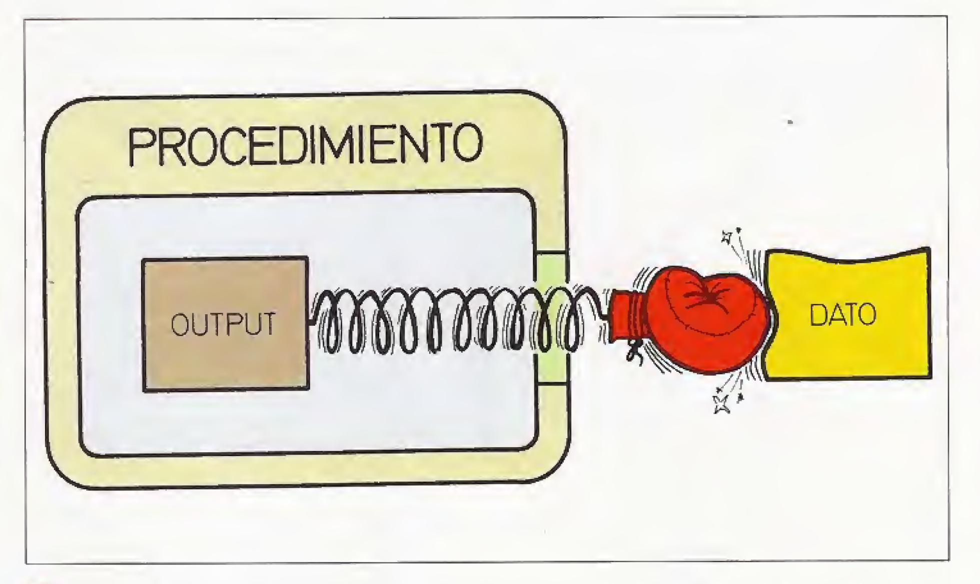
Las colisiones pueden ocurrir al coincidir una tortuga con una determinada porción de la pantalla, o al entrar en contacto dos tortugas. Las distintas combinaciones posibles se codifican por medio de un número. Este se añade a continuación del operador COND para que éste detecte se produce tal evento. Los códigos de colisión dependen de cada dialecto LOGO; en cualquier caso, es posible evitar su empleo gracias a otros dos operadores.

TOUCHING es un operador que devuelve el código de la colisión entre dos tortugas. Su argumento incluye los números identificativos de las tortugas deseadas. Así pues, para detectar una colisión entre las tortugas 1 y 2 basta con explorar el dato de salida de COND TOUCHING 1 2.

Para identificar la coincidencia de posición de una tortuga y un trazo en la pantalla se hace uso de OVER. Así, por ejemplo, la siguiente lista de órdenes mostrará el valor lógico de la colisión de la tortuga 0 con un trazo creado en la pantalla con la tiza número 2:

```
PRINT COND OVER 0 2
```

El dato de salida de COND puede identificarse dentro de una orden IF, para bifurcar hacia acciones distintas. De esta forma, se puede ejecutar una secuencia de instrucciones específica cuando se produzca una colisión.

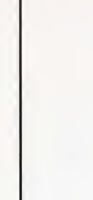



*El comando OUTPUT sirve para canalizar un dato de salida del procedimiento. Dicho procedimiento se convierte, de esta forma, en operador LOGO.*

## Detección instantánea

El problema que subyace en el uso conjunto de IF y COND, dériva de la necesidad de ejecutar dichas órdenes repetidamente. Si la colisión se produce antes o después de chequear la condición, su efecto no se verá reflejado al ejecutar el procedimiento. En definitiva: la colisión sólo se detecta al ejecutarse la cláusula COND. Se puede utilizar otro método para detectar colisiones «en el acto». El comando implicado es WHEN. Los dos parámetros de entrada de WHEN son el código de colisión y una lista de instrucciones. Las instrucciones de la lista se ejecutarán *inmediatamente,* en cuanto se produzca la colisión.

Veamos un ejemplo ilustrativo de ambos métodos de detección.



*Un procedimiento recursivo es aquel que se llama así mismo. Ello equivale a disponer de un número ilimitado de procedimientos análogos.*

```
TO PASEO
TELL 0 1
ST
CS
PU
ASK 0[FORWARD 50 SETSP 50]
COLISION
END
```

```
TO COLISION
IF COND TOUCHING 0 1 [ASK 0 [RIGHT
180 FORWARD 10]]
COLISION
END
```

El primer procedimiento PASEO llama a las tortugas 0 y 1. La número 0 se moverá a una velocidad de 10. A su vez, el procedimiento COLISION se encarga de detectar el choque entre ambas tortugas. Cuando esto sucede, la tortuga móvil (la número 0) da media vuelta.

Este mismo ejemplo es programable con la ayuda de WHEN:

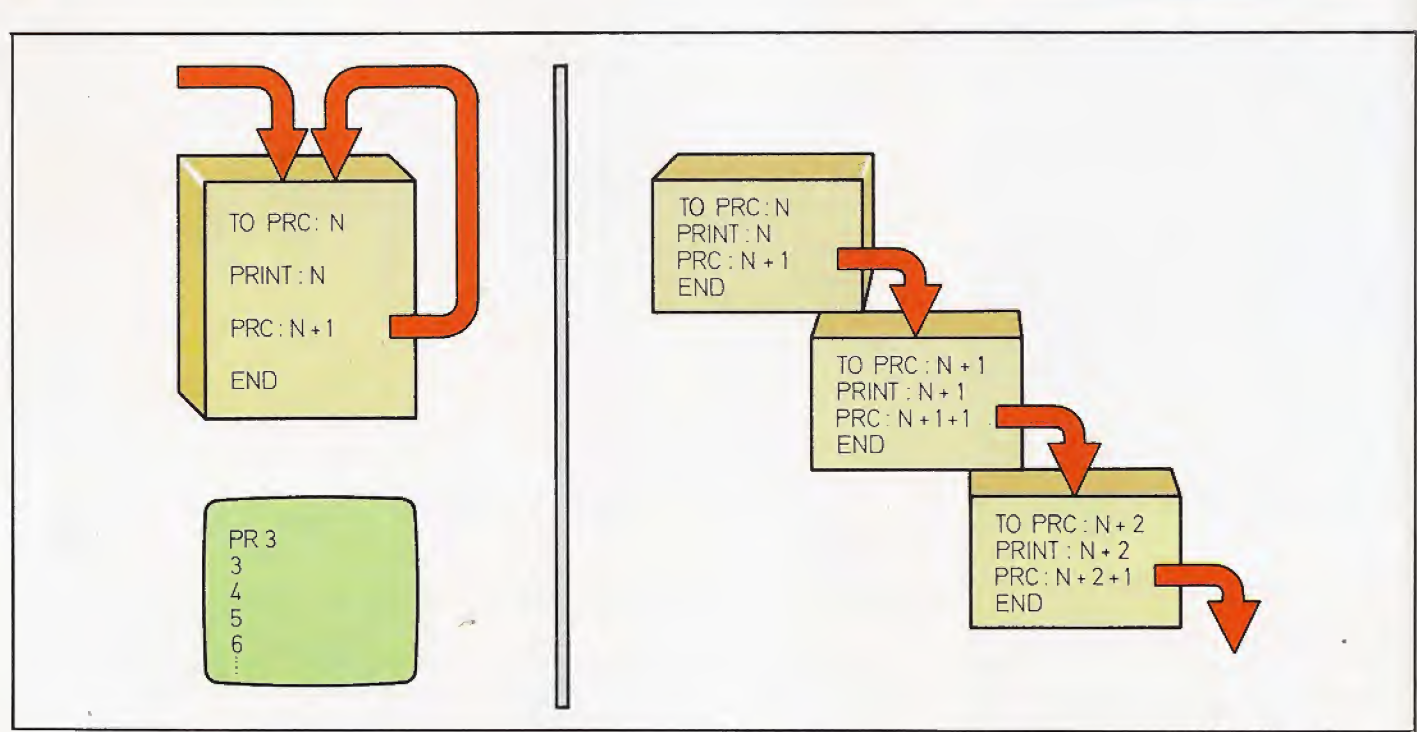
```
TO PASEO1
TELL 0 1
ST
CS
PU
ASK 0 [FORWARD 50 SETSP 50]
WHEN TOUCHING 0 1 [ASK 0 [RIGHT 180]]
END
```

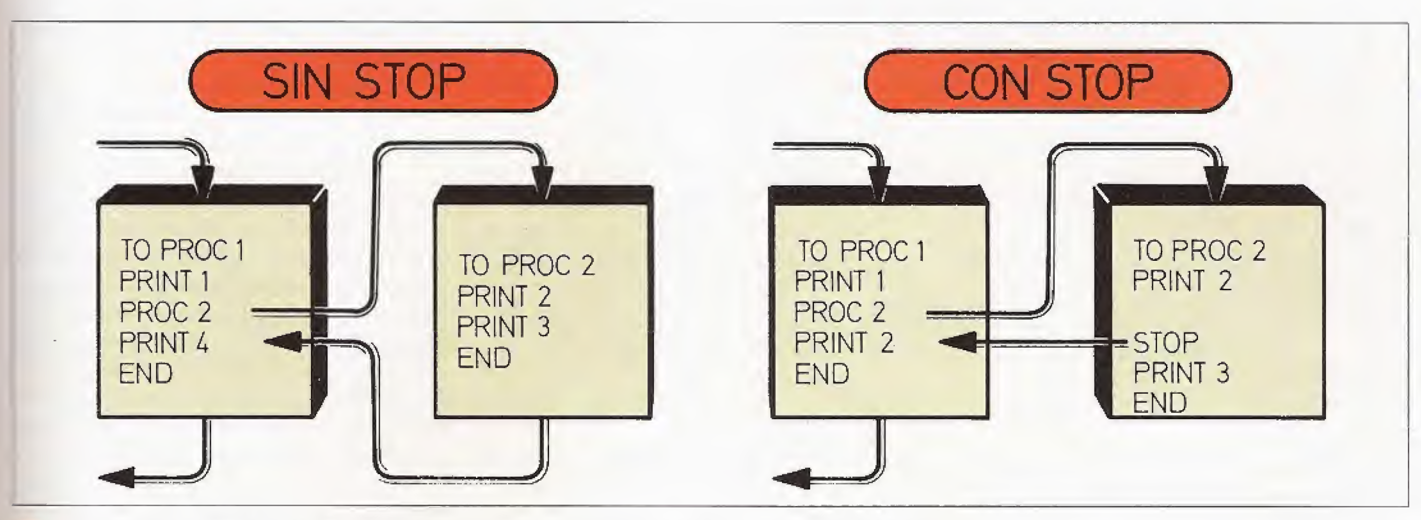
## La recursividad

Cualquier lista de instrucciones LOGO debe estar encabezada por un comando. Si lo que se desea es que esa lista dé como resultado un dato de salida, es evidente que éste debe de ser canalizado adecuadamente. En este segundo caso, la lista actúa como un operador.

Existe una orden LOGO especializada en «encauzar» datos de salida. La palabra LOGO que la identifica es OUTPUT.

```
TO RESTA :M :N
OUTPUT :M—:N
END
```






*El uso del comando STOP permite abandonar un procedimiento, antes de que éste llegue al final del mismo delimitado por la orden END.*

El procedimiento RESTA opera la sustracción en modo prefijo. En su interior, el comando OUTPUT es quien se encarga de que el resultado aparezca como dato de salida del procedimiento.

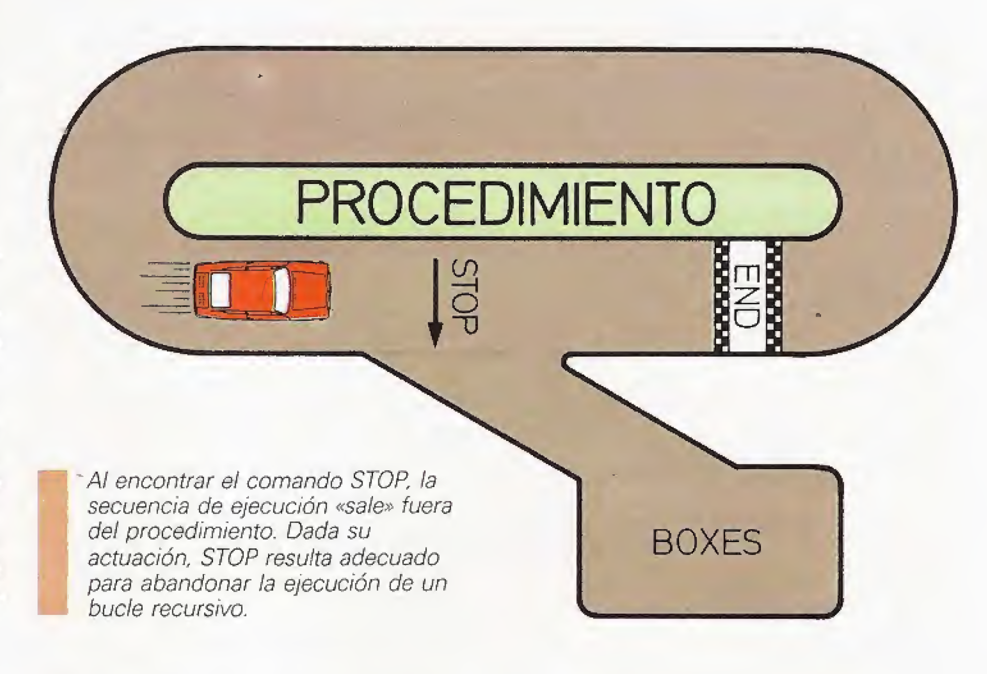
De este modo es posible construir procedimientos que actúen como operadores; claro está, estos procedimientos deben ir precedidos por un comando.

Además de permitir la creación de nuevos operadores, esta característica permite, también, hacer uso de un nuevo concepto: la recursividad. Este término, muy utilizado en teoría matemática, se refiere a la posibilidad de incluir lo definido dentro de la definición. Para ilustrar este nuevo concepto, construiremos un procedimiento cuyo cometido es definir los números pares de forma recursiva. El punto de partida se halla en la siguiente constatación:

El número N es par si:
- N es igual a 2, o
- N—2 es par.

Sin lugar a dudas, este método para la detección de números pares resulta bastante engorroso para utilizarlo «a mano». No obstante, el ordenador puede ponerlo en práctica muy fácilmente:

```
TO PAR :NU
IF :NU=2 [OUTPUT "TRUE] [IF PAR :NU—2
[OUTPUT "TRUE]]
END
```

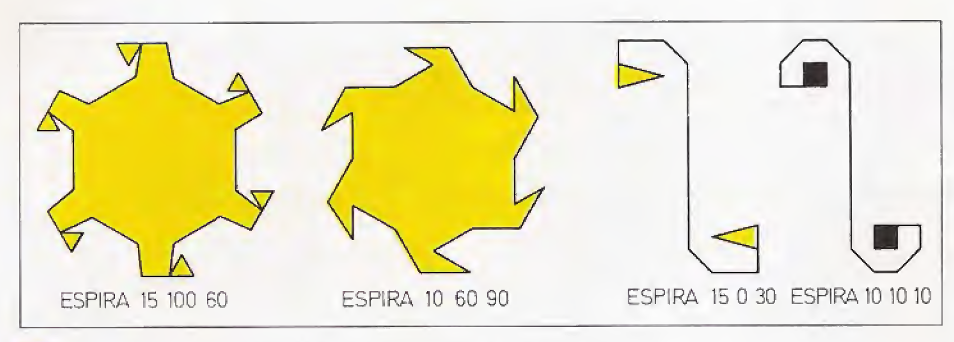



*Al encontrar el comando STOP, la secuencia de ejecución «sale» fuera del procedimiento. Dada su actuación, STOP resulta adecuado para abandonar la ejecución de un bucle recursivo.*

El procedimiento empieza comparando el dato de entrada con el número 2. Si no son iguales, se procede a efectuar la siguiente comparación; en esta segunda comparación se vuelve a hacer uso de PAR, aunque ahora el dato comparado con el 2 es N—2. En el supuesto de que el número N fuera 6, la ejecución coincidirá con la que se expone a continuación:

```
PAR 6
     IF 1)6≠2
     IF 2) PAR 4
          IF 2.1)4≠2
          IF 2.2) PAR 2
               IF 2.21)2=2
          TRUE
     TRUE
TRUE
```

Tal como se observa, el procedimiento PAR se llama a sí mismo en dos ocasiones. Es posible que el ejemplo anterior resulte de ardua asimilación. Realmente, el concepto de recursividad exi-




*Resultado de la ejecución del procedimiento ESPIRA con distintos parámetros de entrada.*

ge un gran poder de abstracción. En todo caso, su comprensión puede verse facilitada con la ayuda de la tortuga.

El siguiente ejemplo conjuga la recursividad con el tratamiento gráfico. Se trata, exactamente, de dibujar un polígono espiral. Ello puede conseguirse como sigue:

1. Avanzar la tortuga X posiciones (por ejemplo, 5).
2. Girar un determinado ángulo (por ejemplo, 90 grados).
3. Avanzar 2+X posiciones.
4. Girar el mismo ángulo señalado en el punto 2.
5. Avanzar 4+X posiciones.

...

Las acciones de «avanzar» y «girar» pueden agruparse en una lista. Por ejemplo:

```
[FORWARD :X LEFT 90]
```

La referida lista de instrucciones debe ejecutarse repetidamente alterando tan sólo el valor de :X.

Apelando al concepto de recursividad, el método apropiado se resumirá en los siguientes puntos:

a) Ejecutar la lista para un valor de X.
b) Repetir el método con un valor de X superior en 2 unidades al anterior.

El segundo enunciado implica volver al punto «a» con un valor igual a 2 + X. Este ciclo se repetirá en sucesivas ocasiones, con valores crecientes de X en dos unidades por cada recorrido. La puesta en práctica de esta secuencia equivale a crear un polígono espiral en modo recursivo.

El procedimiento LOGO adecuado, presentará el siguiente aspecto:

```
TO ESPI :X
FORWARD :X
LEFT 90
ESPI 2+:X
END
```

El valor inicial de la variable X puede ser fijado a voluntad; de acuerdo a su magnitud se obtendrán espirales de distinto tamaño. Por supuesto, también es posible introducir el ángulo a modo de variable. E incluso puede escribirse una variante del anterior procedimiento que aporte una variación en el incremento sucesivo del valor de X:

```
TO ESPIRA :X :A :I
FORWARD :X
LEFT :A
ESPIRA :X :A+I :I
END
```

Este último ejemplo permite las más variadas figuras, puesto que altera los

| OPERADORES LOGICOS AND, OR Y NOT | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | AND | | | OR | | NOT |
| DATO 1 | DATO 2 | SALIDA | DATO 1 | DATO 2 | SALIDA | DATO | SALIDA |
| TRUE<br>TRUE<br>FALSE<br>FALSE | TRUE<br>FALSE<br>TRUE<br>FALSE | TRUE<br>FALSE<br>FALSE<br>FALSE | TRUE<br>TRUE<br>FALSE<br>FALSE | TRUE<br>FALSE<br>TRUE<br>FALSE | TRUE<br>TRUE<br>TRUE<br>FALSE | TRUE<br>FALSE | FALSE<br>TRUE |

| TABLA DE COMANDOS LOGO (1) | | |
|---|---|---|
| Instrucción | Cometido | Operador/ Comando |
| SUM <N1> <N2> | Realiza la suma de los datos de entrada (prefijo) | Operador |
| <N1> + <N2> | Realiza la suma de los datos de entrada (infijo) | Operador |
| PRODUCT <N1> <N2> | Obtiene el producto de los datos de entrada (prefijo) | Operador |
| <N1> * <N2> | Obtiene el producto de los datos de entrada (infijo) | Operador |
| <N1> - <N2> | Opera la resta de los datos entre los que se encuentra (infijo) | Operador |
| <N1> / <N2> | Divide los datos que acompañan al operador (infijo) | Operador |
| SQRT <N1> | Halla la raíz cuadrada del dato de entrada | Operador |
| SIN <N1> | Calcula el seno del dato de entrada | Operador |
| COS <N1> | Calcula el coseno del dato de entrada | Operador |
| Nota: <N1> y <N2> representan datos numéricos. | | |

valores iniciales de ángulo, longitud e incremento.

## Limitar la recursividad

En los ejemplos anteriores, la ejecución del procedimiento no tiene fin. El procedimiento continúa «llamándose» a sí mismo indefinidamente, y sólo se detendrá la ejecución al emitirse un mensaje de error: al exceder la variable del valor límite permitido. Desde luego, la interrupción también puede derivar de una orden al efecto dada por el propio usuario desde el teclado.

Normalmente, es necesario «limitar» la acción de un procedimiento recursivo. Ello es posible gracias al comando STOP. Su cometido se entenderá con mayor claridad dejando al margen, por el momento, el tema de la recursividad.

```
TO PROC1
PRINT 1
PROC2
PRINT 4
END

TO PROC2
PRINT 2
PRINT 3
END
```

El primer procedimiento (PROC1), tras escribir el dato 1, «llama» al segundo (PROC2). Este último mostrará en la pantalla los números 2 y 3; tras ello, devolverá el control a PROC1. Acto seguido, proseguirá la ejecución de PROC1 escribiendo la cifra 4.

Veamos, en definitiva, cuál es el resultado de su ejecución:

```
PROC1

1
2
3
4
```

**TABLA DE COMANDOS LOGO (2)**

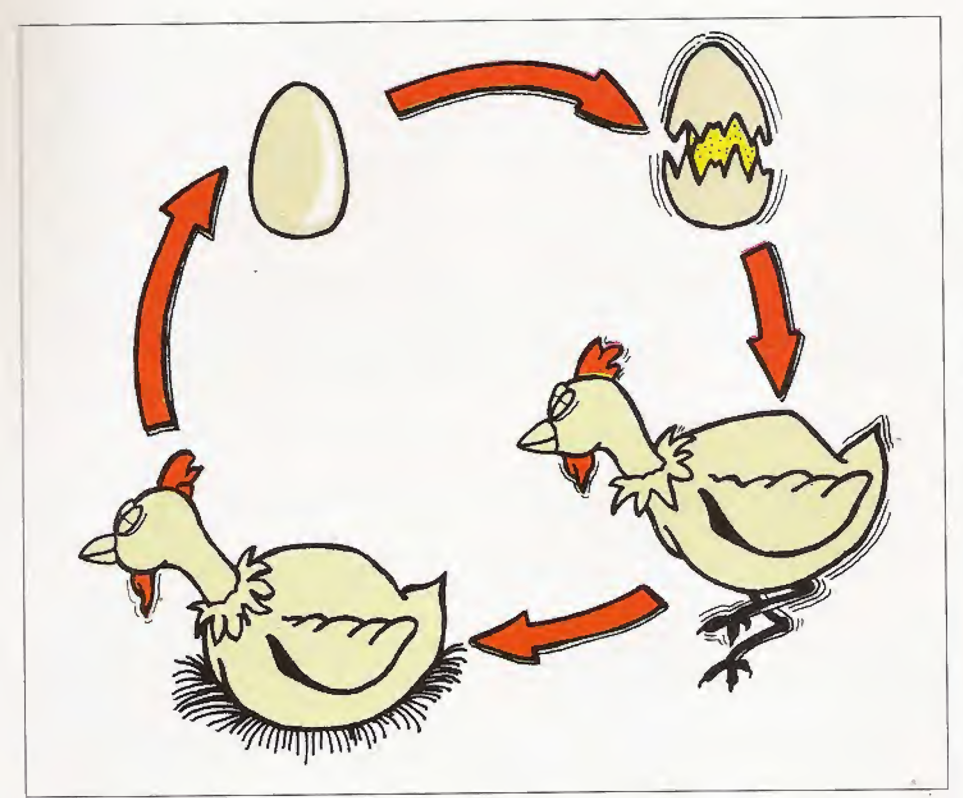
| Instrucción | Cometido | Operador/ Comando |
|---|---|---|
| RANDOM <N1> | Genera un número entero aleatorio inferior al valor dado | Operador |
| RERANDOM | Recuerda la última secuencia generada de números aleatorios | Comando |
| <N1> > <N2> | Entrega el dato lógico TRUE si N1 es mayor que N2 | Operador |
| <N1> < <N2> | Proporciona el dato lógico TRUE si N1 es menor que N2 | Operador |
| EQUALP <N1> <N2> | Proporciona el dato lógico TRUE si N1 y N2 son iguales (prefijo) | Operador |
| <N1> = <N2> | Proporciona el dato lógico TRUE si N1 y N2 son iguales (infijo) | Operador |
| AND <L1> <L2> | Realiza la función lógica AND (Y) entre los datos de entrada | Operador |
| OR <L1> <L2> | Realiza la función lógica OR (O) entre los datos de entrada | Operador |
| NOT <L1> | Entrega el valor lógico contrario al dato de entrada (NO) | Operador |
| Nota: <N1> y <N2> representan datos numéricos. | | |

**TABLA DE COMANDOS LOGO (3)**

| Instrucción | Cometido | Operador/ Comando |
|---|---|---|
| REPEAT <n> <lista> | Repite n veces las órdenes indicadas en la lista | Comando |
| IF <cond> <lista 1> <lista 2> | Si la condición es cierta se ejecuta la lista 1, en caso contrario se ejecuta la lista 2. | Depende de la lista |
| COND <n> | Devuelve el valor TRUE si la colisión número n se cumple | Operador |
| TOUCHING <T1> <T2> | Devuelve el número identificativo de la colisión entre las tortugas número T1 y T2 | Operador |
| OVER <Tor> <Tiza> | Devuelve el código de colisión entre la tortuga número T y la tiza cuyo número se indica | Operador |
| RUN <lista> | Ejecuta las órdenes indicadas en la lista | Depende de la lista |
| WHEN <n> <lista> | Ejecuta la lista en el mismo instante en que se produzca la colisión número n | Depende de la lista |
| OUTPUT <dato> | Coloca su dato de entrada como dato de salida del procedimiento en el que se encuentra | Comando |
| STOP | Interrumpe la ejecución de un subprocedimiento para volver al procedimiento que lo llamó | Comando |
| Nota: <n>: Dato numérico (número de veces o número de colisión). <lista>, <lista1> y <lista2>: Lista de instrucciones. <Tor>, <T1> y <T2>: Número de tortuga. <Tiza>: Número de tiza. <cond>: Condición que genera un número de tipo lógico. | | |

Modifiquemos ahora, ligeramente, el subprocedimiento PROC2. Su nuevo aspecto es el siguiente:

```
TO PROC2
PRINT 2
STOP
PRINT 3
END
```



*La recursividad hace que la secuencia de control entre en un círculo cerrado; una vez en su interior, se diluye el significado de entrada y salida o de principio y final de un procedimiento... ¿Qué fue antes, el huevo o la gallina?*

La alteración queda plasmada por la presencia de un comando STOP entre las dos órdenes de escritura. Al ejecutar de nuevo el procedimiento PROC1, la pantalla reflejará el efecto de la modificación operada:

```
PROC1

1
2
4
```

Como puede apreciarse, la instrucción PRINT 3 no se ha ejecutado. En cambio, sí aparece escrito el 4.

La acción de STOP es devolver el control del procedimiento que efectuó la llamada al procedimiento en el que se encuentra dicho comando. En otros términos: STOP retorna la ejecución al nivel inmediatamente superior (entendiendo que los procedimientos que «llaman» se sitúan en un nivel superior a los «llamados»). De no existir un nivel superior —el procedimiento no ha sido llamado por otro—, aparecerá el mensaje de error YOU'RE AT TOP LEVEL (está usted en el nivel superior).

Este comando va a resultar muy útil en procedimientos recursivos. Al «llamarse» a sí mismos, este tipo de procedimientos van recorriendo niveles cada vez inferiores.

En los ejemplos anteriores se descendía de nivel indefinidamente. Ahora, con la ayuda de STOP, se puede volver a subir hasta llegar al nivel de partida. he aquí un ejemplo sencillo en el que se conjuga la recursividad con el uso del comando STOP.

```
CUENTA :N
IF :N=0 [PRINT "CERO STOP]
PRINT :N
CUENTA :N—1
PRINT :N
END
```

El comando STOP actúa deteniendo el proceso recursivo. Una vez que se ha regresado al nivel inmediatamente superior, ya no se «bajará» de nuevo. Ello se debe a que, después de llamar al nivel inferior (cuarta línea), el resto del procedimiento no vuelve a efectuar una nueva llamada. El resultado de la ejecución adoptará el siguiente aspecto:

```
CUENTA 4
4
3
2
1
CERO
1
2
3
4
```

El primer PRINT es el encargado de escribir los números en orden descendente. Al llegar a 0, se escribe «cero» y se interrumpe la recursividad. La sucesiva vuelta al nivel superior produce la cuenta ascendente, cuyos valores llegan a la pantalla por medio del segundo PRINT. La siguiente «radiografía» de la ejecución ilustra el desarrollo del proceso. Para no prolongar excesivamente el ejemplo, se ha tomado el número 2 como dato de partida.

| línea | instrucción |
|---|---|
| 1 | CUENTA :N(2) |
| 2 | IF :N=0... (2 <> 0) |
| 3 | PRINT :N (2) |
| 4 | PRINT :N — 1 (1) |
| 1 | CUENTA :N (1) |
| 2 | IF:N=0... (1 <> 0) |
| 3 | PRINT :N (1) |
| 4 | CUENTA :N — 1 (0) |
| 1 | CUENTA :N (0) |
| 2 | IF :N=0 [PRINT "CERO STOP] |
| 5 | PRINT :N (1) |
| 6 | END |
| 5 | PRINT :N (2) |
| 6 | END |

# LOGO (y 5)

## El diálogo con los periféricos



Hasta el momento, el diálogo con el ordenador en lenguaje LOGO se reducía a introducir las órdenes oportunas a través del teclado. El programador comunicaba, de esta forma, sus deseos a la máquina. El resultado podía verse reflejado en la pantalla de dos formas: por medio de trazos creados por la tortuga o en forma de palabras (o listas) escritas en la pantalla por medio de instrucciones PRINT.

En este capítulo se amplían los métodos de comunicación con el aparato; desde el control total de textos en pantalla, hasta la posibilidad de introducir datos durante la ejecución de un procedimiento. También se repasarán los medios existentes para el almacenamiento de los programas

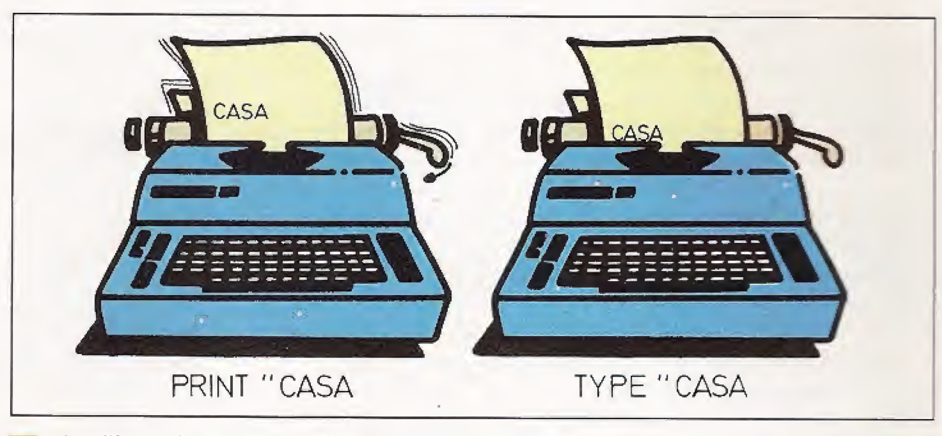
### Manejo del texto

El comando que se ha utilizado hasta ahora para la visualización de textos es PRINT. Este admite un dato de entrada que puede ser una palabra, una lista o un dato numérico. Por ejemplo:

```
PRINT "CRISTINA
CRISTINA

PRINT 43
43

PRINT [1 2 3 4]
1 2 3 4
```




*La diferencia entre los comandos PRINT y TYPE reside en que el primero provoca un salto a la siguiente línea tras escribir el texto indicado.*

Cada vez que se ejecuta una orden PRINT, se salta a una nueva línea de pantalla. El siguiente procedimiento ilustra dicha particularidad.

```
TO LEER
PRINT "JUAN
PRINT "LEE
PRINT "LIBROS
END
```

Al ejecutar este procedimiento se obtendrá la siguiente respuesta en la pantalla:

```
JUAN
LEE
LIBROS
```

Para visualizar todas las palabras en una misma línea, habría que agruparlas dentro de una lista; aunque ello no siempre es posible. Por lo demás, una vez escrita una parte del texto, no cabe la posibilidad de proseguir la escritura a continuación de la última palabra visualizada.

```
TO LEER :BUENO
PRINT [JUAN LEE]
IF:BUENO= "LIBROS [PRINT :BUENO]
END
```

En el ejemplo, la palabra LIBROS quedará siempre una línea por debajo de las restantes.

```
LEER "LIBROS
JUAN LEE
LIBROS
```

SETCURSOR [10 5] TYPE "PEPE

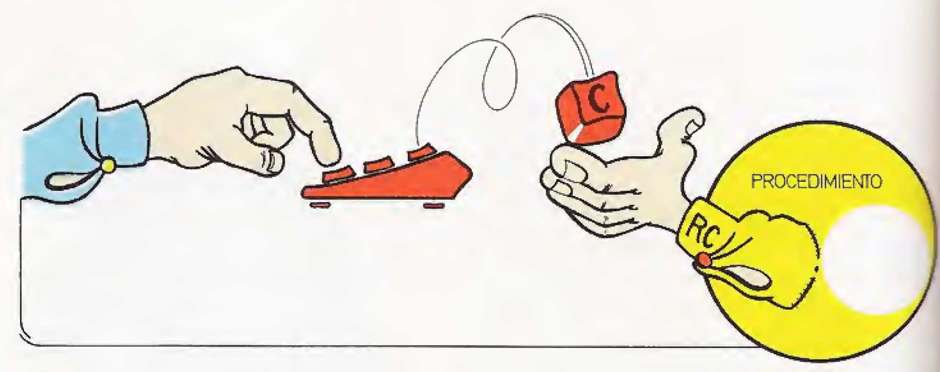
0 10
5 PEPE?

*La orden SETCURSOR permite situar la visualización en cualquier punto de la pantalla, sin más que especificar las coordenadas en su argumento.*

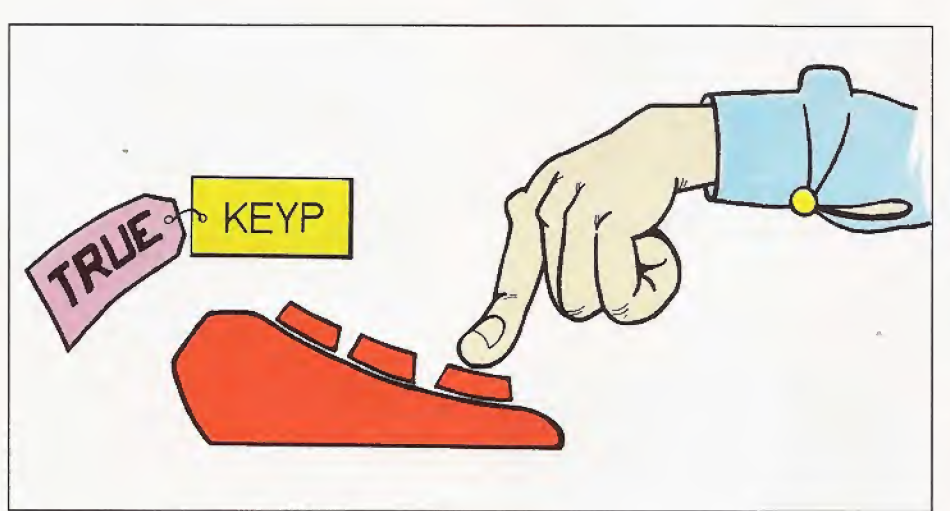
Para evitar tal circunstancia, hay que recurrir a un comando que no ejecute un salto de línea después de realizar la escritura. Este comando es TYPE. El ejemplo precedente puede modificarse incluyendo el nuevo comando:

```
TO LEER :BUENO
TYPE [JUAN LEE]
IF :BUENO= "LIBROS [PRINT :BUENO]
END
```

Ahora, su ejecución conducirá a la siguiente presentación en pantalla:




*El operador RC se utiliza para recoger en un procedimiento el carácter introducido a través del teclado.*




*Recurriendo a la orden KEYP es posible detectar cualquier acción que se produzca sobre el teclado. Dicho operador responde con TRUE (cierto) cuando se acciona una tecla.*

```
LEER "LIBROS
JUAN LEELIBROS
```

El resultado no es del todo satisfactorio, ya que sería aconsejable dejar un espacio en blanco entre las dos últimas palabras. Para visualizar dicho espacio en blanco, hay que utilizar el signo \ (backslash). Con esta nueva modificación, nuestro reiterado ejemplo quedará ya en perfecto orden:

```
TO LEER :BUENO
TYPE [JUAN LEE]
TYPE" \
IF :BUENO= "LIBROS [PRINT :BUENO]
END
```

visualizando ya el mensaje en la forma deseada:

```
LEER "LIBROS
JUAN LEE LIBROS
```

En definitiva, TYPE permite encadenar datos en la misma línea de presentación. Tanto TYPE como PRINT admiten más de un dato de entrada, empleando el método de encerrar los datos y el comando entre paréntesis. Tal posibilidad es la que se ilustra a continuación:

```
(PRINT "JUAN "LEE "LIBROS)
JUAN
LEE
LIBROS

(TYPE "JUAN "LEE "LIBROS)
JUANLEELIBROS
```

Existe otro comando relacionado con la visualización de datos. Este es SHOW. Su función es idéntica a la de PRINT, con una única salvedad en la presentación de listas. Veamos un ejemplo.

```
PRINT [DIA SEIS DE] PRINT "MAYO
DIA SEIS DE
MAYO

SHOW [DIA SEIS DE] SHOW "MAYO
[DIA SEIS DE]
MAYO
```

Por último, cabe mencionar al comando SETCURSOR, que permite elegir el punto de la pantalla en el que debe empezar la próxima impresión. El punto se especifica por medio de una lista de números: el primero representa la columna y el segundo la fila adecuadas. Los valores máximos de ambos parámetros dependerán de la resolución de la pantalla, en modo texto, característica de cada ordenador.

## Recogida de datos

Es posible introducir datos durante la ejecución de un procedimiento. Para ello es necesario instruir al ordenador oportunamente. La introducción manual de datos puede realizarse por medio del teclado o a través de un periférico especial: *joystick* (palanca de juegos) o *paddle* (potenciómetro de control).

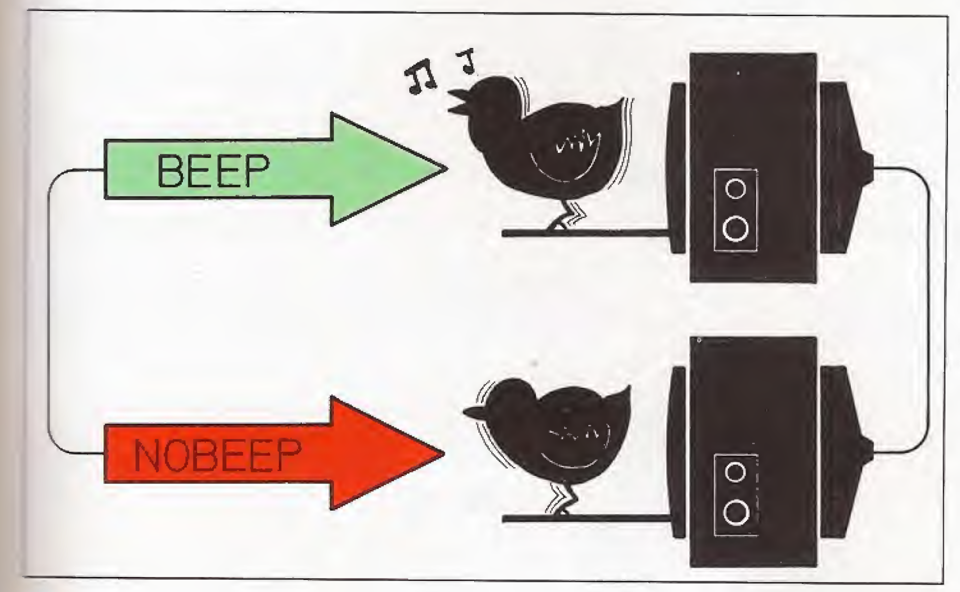
En el primer caso se dispone de dos operadores adecuados para leer datos del teclado: RC y RL. RC (de Read Character, leer carácter) interrumpe la ejecución del procedimiento hasta que se pulsa una tecla. Una vez accionada, el carácter correspondiente a la tecla pulsada aparece como dato de salida de RC, continuando de inmediato la ejecución del procedimiento. Cabe señalar que dicho carácter no es visualizado en la pantalla.

La misión de RL es similar, si bien éste operador actúa con listas (Read List, leer lista). Con RL es necesario pulsar la tecla RETURN para indicar al ordenador que ha finalizado la introducción del dato. Los datos tecleados tras la ejecución de RL sí aparecen en pantalla.

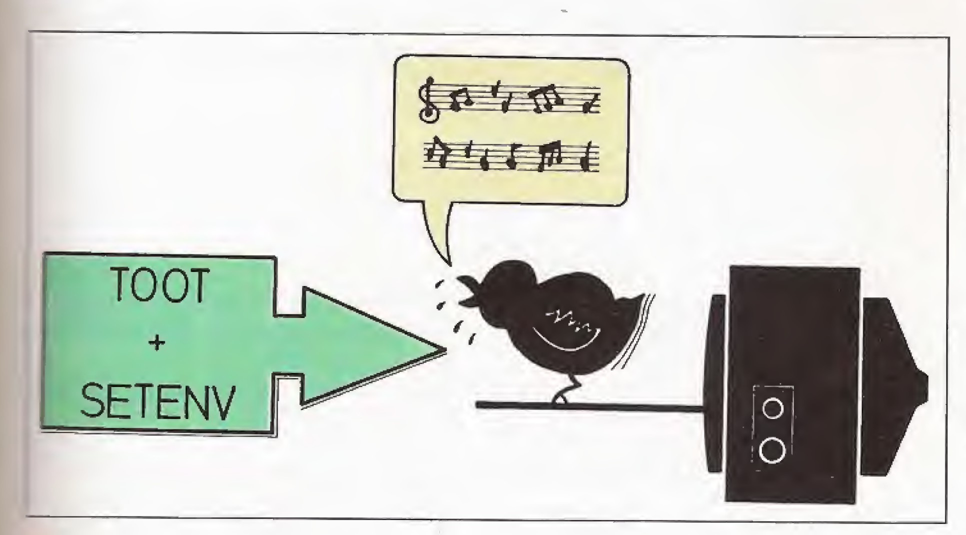
Existe un tercer operador especializado en detectar cualquier acción sobre el teclado: KEYP (de key, tecla). Proporciona un dato de tipo lógico (TRUE) si, en efecto, se ha pulsado una tecla. A diferencia con los anteriores, KEYP no detiene la ejecución del procedimiento en el que se encuentra. El siguiente ejemplo hace uso de KEYP para detectar la pulsación de una tecla y de RC para identificar cuál es la tecla accionada. El objetivo es mover a la tortuga en las direcciones vertical y horizontal con las teclas W (arriba), Z (abajo), A (izquierda) y S (derecha).

```
TO MOVIL
IF KEYP [ANDA RC]
MOVIL
END

TO ANDA :HACIA
IF :HACIA="W [SETH 0]
IF :HACIA="S [SETH 90]
IF :HACIA="Z [SETH 180]
IF :HACIA="A [SETH 270]
FORDWARD 15
END
```




*Los comandos de sonido más elementales y frecuentes en cualquier versión de LOGO son BEEP y NOBEEP. Su única misión es la de conectar o desconectar el zumbador.*




*Algunas versiones de LOGO permiten generar sonidos complejos, por efecto de la acción conjunta de dos comandos: TOOT y SETENV.*

## Joysticks y paddles

Por lo que respecta a la recogida de datos procedentes de periféricos especiales, la mayor importancia recae en las órdenes destinadas a joysticks y paddles, los dos periféricos especiales más comunes.

Los operadores implicados son JOY y JOYB para el joystick, y PADDLE y PADDLEB para el paddle.

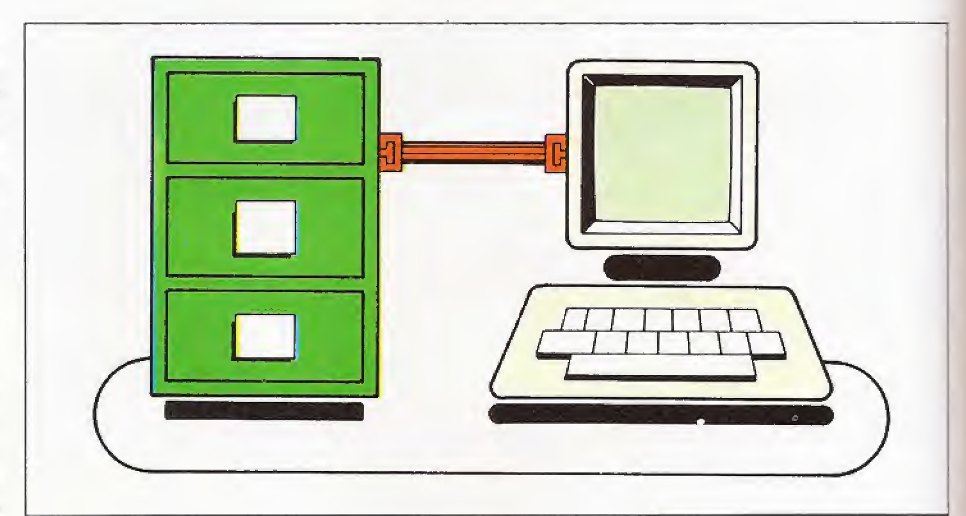
JOY devuelve un número indicativo de la dirección en la que se ha desplazado la palanca del joystick,. Estos valores no son comunes para todos los ordenadores, de ahí que, en cada caso, haya que consultar el manual propio de cada aparato. La misión de JOYB es detectar la pulsación del botón de disparo del joystick, entregando el dato TRUE si ocurre tal situación.

*El lenguaje LOGO dispone de funciones específicas para el control de periféricos especiales como, por ejemplo, mandos de juego y tabletas.*

El operador PADDLE obtiene un dato numérico comprendido entre cero y un número positivo máximo. Este dato representa el ángulo girado en el botón de mando del paddle. La función PADDLEB es idéntica a la de JOYB, con la salvedad de que en este caso el botón es el de paddle.

## Sonidos

El uso de las capacidades sonoras del ordenador por medio de órdenes LOGO no está, en absoluto, estandarizado. Existen muy diversas alternativas que van desde la versión más simple, que sólo dispone de BEEP (inicia un zumbido) y NOBEEP (cancela el zumbido), hasta otras más complejas y con un mayor surtido de órdenes al efecto. En esta última categoría, suelen estar presentes los comandos TOOT y SETENV. El primero admite datos numéricos de entrada para establecer la frecuencia y el volumen del sonido. A su vez, SETENV controla la envolvente para obtener efectos sonoros más complejos. Si se dispone de equipos con más de un canal sonoro, se especificará también el canal deseado. Es obvio que dada la disparidad de dialectos LOGO en este punto, resulta imprescindible consultar el manual específico de cada equipo.



*En el lenguaje LOGO, el almacenamiento de información en las memorias de masa se realiza en forma de archivos.*

## La periferia del ordenador

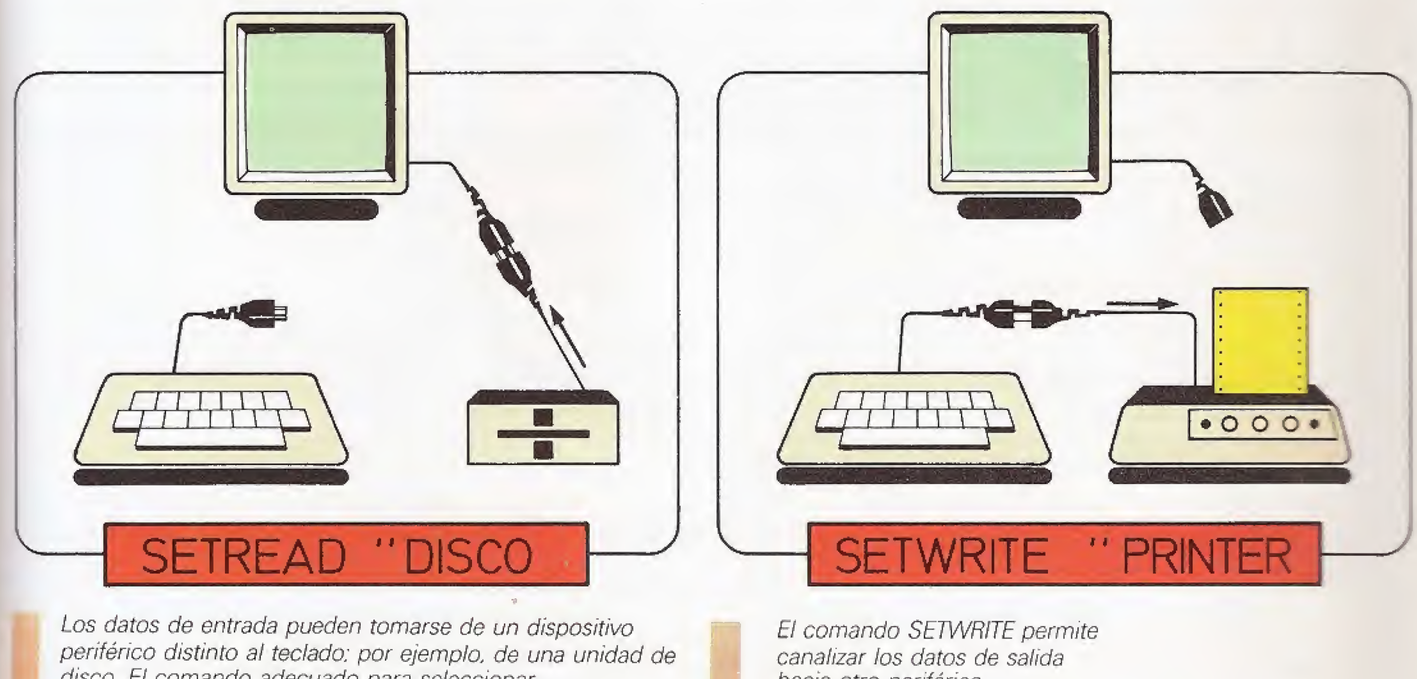
El entorno que rodea al ordenador incluye todo un abanico de dispositivos periféricos. Entre ellos se encuentran las impresoras, unidades de disco, de casete, etc. En los dispositivos dedicados al almacenamiento masivo (cintas y discos) se suele agrupar la información en «archivos».

Un archivo es un conjunto homogéneo de datos manipulable bajo un nombre genérico. Los archivos son, pues, estructuras de datos.

El primer ejemplo surge con el deseo de conservar los procedimientos crea-

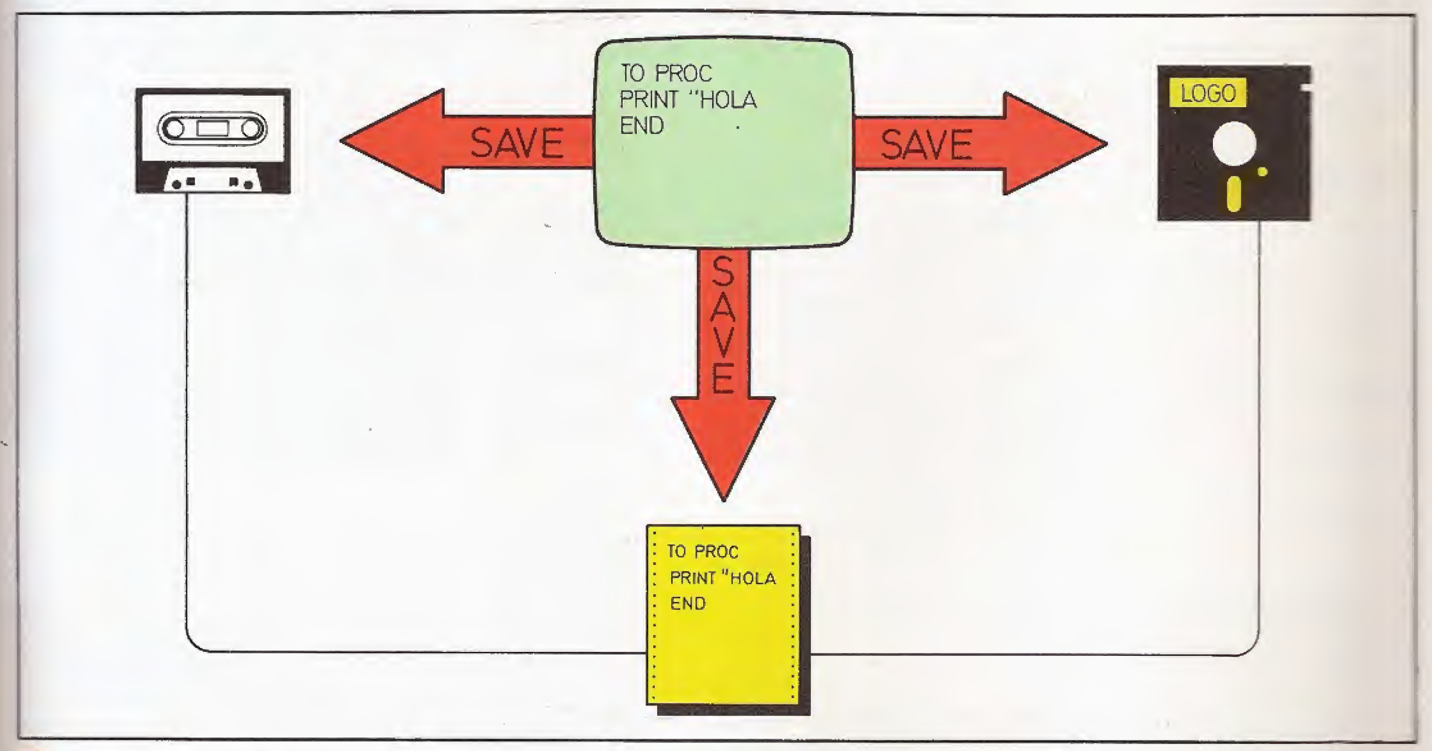
dos en una sesión de trabajo. Es sabido que al desconectar el aparato el contenido de su memoria se pierde. Para evitar la repetición del mismo trabajo, hay que hacer uso de los periféricos de almacenamiento permanente. Los más populares son la cinta y el disco magnético. Con ellos es posible preservar la información contenida en el espacio de trabajo.

El comando que permite archivar la información es SAVE. Este necesita un

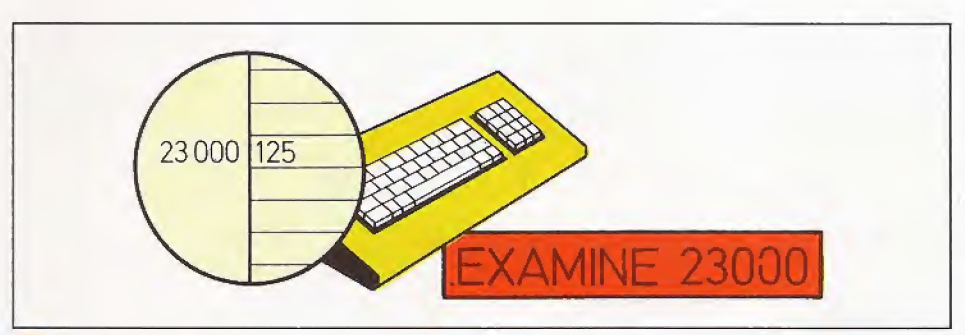



*Los datos de entrada pueden tomarse de un dispositivo periférico distinto al teclado; por ejemplo, de una unidad de disco. El comando adecuado para seleccionar tal posibilidad en SETREAD.*

*El comando SETWRITE permite canalizar los datos de salida hacia otro periférico distinto de la pantalla.*




*El comando SAVE es la herramienta LOGO adecuada para ordenar la transferencia y almacenamiento de información en casete, disco e incluso papel (por medio de la impresora conectada al equipo).*




*La función "EXAMINE" permite entrar en la intimidad de la memoria central del ordenador, revela el contenido de la posición de memoria que se indique como parámetro.*

dato de entrada que será una palabra identificativa del archivo creado. En un mismo soporte (cinta en casete o disco) se pueden guardar varios archivos. Este es el motivo por el cual es preciso diferenciarlos con distintos nombres.

En el caso de que exista más de un dispositivo de almacenamiento conectado al ordenador, es preciso indicar cuál de ellos debe memorizar el archivo. La identificación de los periféricos suele diferir de uno a otro equipo; aunque, con frecuencia, se suele recurrir a las siguientes letras: C (casete), D (unidad de disco) y P (impresora).

Si el periférico elegido es la impresora, se obtendrá una copia en papel del contenido del espacio de trabajo.

El comando que realiza la operación contraria es LOAD. Este carga en memoria el archivo especificado por su dato de entrada. Así, un espacio de trabajo grabado con SAVE "TORTU1, se puede recuperar con LOAD "TORTU1.

El comando LOAD no puede ser utilizado cuando el periférico elegido es la impresora. Es obvio que este dispositivo acepta datos, aunque no los devuelve al ordenador.

## Manejo de archivos

El manejo de archivos no se limita al almacenamiento de procedimientos. Es posible guardar y recuperar datos (numéricos, palabras y/o listas). Para ello se recurre a permutar los dispositivos de entrada/salida.

El comando SETWRITE hace que los datos de salida vayan a parar a otro dispositivo distinto de la pantalla. Así, por ejemplo, es posible canalizar los datos hacia la impresora, para obtener resultados fácilmente manejables. El siguiente procedimiento hará que los resultados de ejecutar el procedimiento PRO1 salgan a través de la impresora.

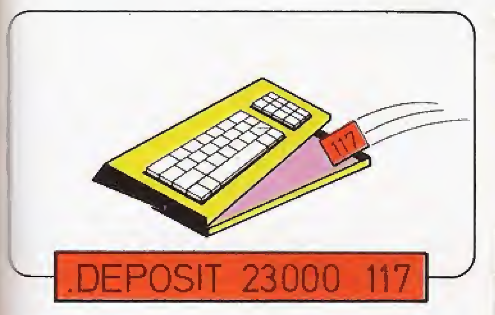
```
TO IMPRE
SETWRITE "P:
PRO1
SETWRITE []
END
```

El periférico de salida puede ser una unidad de almacenamiento, de disco o casete, creándose, de esta forma, archivos de datos no volátiles. De esta forma al ejecutarse una orden PRINT o similar (TYPE, SHOW...) el dato no aparecerá en pantalla, sino que será escrito en el archivo externo. Para volver a activar la pantalla bastará con ejecutar un comando SETWRITE [] (cadena vacía).

**TABLA DE ORDENES LOGO (1)**

| Instrucción | Cometido | Operador/ Comando |
|---|---|---|
| PRINT <dato> | Muestra en pantalla el dato y salta al principio de la línea siguiente | Comando |
| TYPE <dato> | Muestra en pantalla el dato sin saltar de línea | Comando |
| SHOW <dato> | Muestra en pantalla el dato. Si es una lista aparece entre corchetes | Comando |
| SETCURSOR <lista> | Sitúa el siguiente texto a escribir en la posición indicada por la lista de valores | Comando |
| RC | Devuelve el carácter pulsado en el teclado | Operador |
| RL | Devuelve la lista introducida por el teclado | Operador |
| KEYP | Responde con TRUE si en ese instante se pulsa una tecla | Operador |
| JOY[<n>] | Devuelve un dato indicativo de la dirección en la que se desplaza la palanca del joystick | Operador |
| JOYB [<n>] | Devuelve TRUE si se acciona el botón de disparo del joystick | Operador |
| PADDLE [<n>] | Entrega un dato indicativo del giro dado en el botón de control del paddle | Operador |
| PADDLEB [<n>] | Devuelve TRUE si se pulsa el botón de disparo | Operador |
| BEEP | Conecta el zumbador | Comando |
| NOBEEP | Desconecta el zumbador | Comando |

<dato>: Puede ser una palabra, lista o dato numérico. <lista>: Lista formada por dos números. Indican las coordenadas X e Y respectivamente. <n>: Número de dispositivo, en caso de admitir la conexión de más de uno.




*El comando ".DEPOSIT" se encarga de depositar un dato en la posición de memoria que se le indique.*




*Otra de las posibilidades del LOGO se concreta en el examen de las palabras reservadas. Para obtener un listado de todas ellas hay que hacer uso del comando ".PRIMITIVES".*

| TABLA DE ORDENES LOGO (2) | | |
|---|---|---|
| **Instrucción** | **Cometido** | **Operador/Comando** |
| SAVE <arch> | Guarda el espacio de trabajo en el archivo indicado | Comando |
| LOAD <arch> | Recupera un espacio de trabajo del archivo indicado | Comando |
| SETWRITE <arch> | Canaliza los datos de salida hacia el archivo indicado (en lugar de hacia la pantalla) | Comando |
| SETREAD <arch> | Hace que se recojan los datos de entrada del archivo indicado (en lugar de obtenerlos del teclado) | Comando |
| CATALOG | Muestra los nombres de los archivos que residen en el disco | Comando |
| ERF <arch> | Borra del disco el archivo indicado | Comando |
| .PRIMITIVES | Muestra todas las palabras reservadas del LOGO | Comando |
| .EXAMINE <pos> | Devuelve el contenido de una posición de memoria | Operador |
| .DEPOSIT <pos> <N> | Introduce el dato <N> en la posición de memoria especificada | Comando |
| .CALL <pos> | Ejecuta la rutina en código máquina situada en la posición de memoria especificada | Comando |

<arch>: Nombre de archivo; junto al mismo hay que indicar el dispositivo en el que se encuentra. <pos>: Número que identifica una posición de memoria del ordenador. <N>: Dato numérico a situar en una posición de memoria. Ha de ser entero, positivo e inferior a 256.

El método para recuperar un archivo pasa por permutar el teclado con dicho archivo. Esta acción se define indicando el archivo de entrada en un comando SETREAD. Una vez seleccionado el archivo, cada vez que se ejecute una orden RC o RL los datos se tomarán del mismo, en lugar del teclado. La devolución del control al teclado se realiza por medio de la orden SETREAD [].

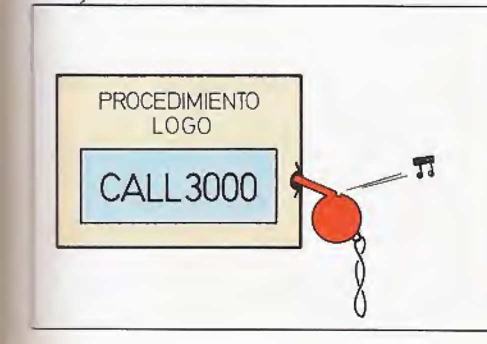
Existen otros dos comandos específicos para la operación con disco magnético. Estos son CATALOG y ERF.

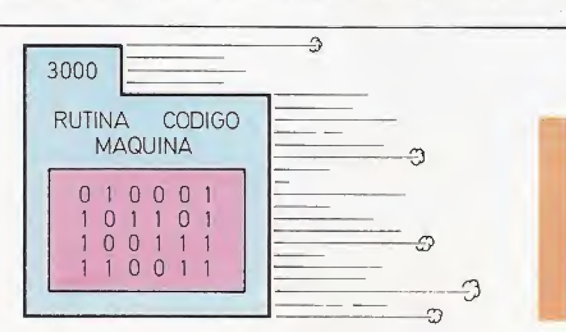
El primero proporciona una tabla que relaciona todos los archivos contenidos en el disco, mientras que ERF borra archivos (ERase File). Este último comando eliminará el archivo cuyo nombre coincida con la palabra incluida como dato de entrada de ERF.

Los dos procedimientos que se relacionan a continuación operan asociados: gestionan el almacenamiento en disco y la emisión de un listado por impresora del procedimiento LOGO que se indique.

```
TO GRABA :P
MAKE "DIS WORD "D: :P
SAVE :DIS
SETREAD :DIS
SETWRITE "P:
LISTADO
SETREAD []
SETWRITE[]
END

TO LISTADO :A
MAKE "L RL
IF EMPTYP :L [STOP]
PRINT :L
LISTADO
END
```




*El cometido de ".CALL" es transferir el control a una rutina en código máquina. Su dato de entrada especifica la posición de memoria a partir de la que está situada la rutina en cuestión.*

| PALABRAS CLAVE DEL LOGO | | | |
|---|---|---|---|
| **Inglés** | **Castellano** | **Inglés** | **Castellano** |
| TS | MODOTEXTO | SETPC | PONCOLOR, PONCL |
| COUNT | CUENTA | PC | COLOR, COLORLAPIZ |
| BUTFIRST | MENOSPRIMERO | SETH | PONRUMBO |
| FIRST | PRIMERO | XCOR | XCOOR |
| BUTLAST | MENOSULTIMO | YCOR | YCOOR |
| LAST | ULTIMO | HEADING | RUMBO |
| FPUT | PONP, COLOCAPRI | POS | DONDE, POSICION |
| LPUT | PONU, COLOCAULT | SETSP | PONVELOCIDAD |
| SE | ORACION, FRASE | SPEED | VELOCIDAD |
| NUMBERP | NUMERO? | WINDOW | VENTANA |
| LISTP | LISTA? | WRAP | ENROLLA |
| WORDP | PALABRA? | TELL | DECIR |
| WORD | PALABRA | WHO | QUIEN |
| EMPTYP | VACIA? | EACH | CADA |
| MEMBERP | PERTENECE? | ROUND | REDONDEA |
| EQUALP | IGUAL? | INT | ENT |
| CHAR | ESCRIBECARACTER, CARACTER | REMAINDER | RESTO |
| | | RANDOM | AZAR |
| ASCII | NUMCARACTER, ASCII | SUM | SUMA |
| MAKE | HACER, ASIGNA | PRODUCT | PRODUCTO |
| NAMEP | NOMBRE? | SIN | SENO |
| THING | COSA, ITEM | COS | COSENO |
| EDIT | EDITA, REVISAR | SQRT | RAIZCUADRADA |
| EDNS | EDNS | QUOTIENT | COCIENTE |
| TO | PARA | TRUE | CIERTO |
| END | FIN | FALSE | FALSO |
| ERNS | SUPNS | AND | AMBOS, Y |
| ERALL | SUPTODO | OR | UNOUOTRO, O |
| ERPS | SUPPS | NOT | NO |
| ERASE | SUPRIME | IF | SI |
| ERN | SUPN | STOP | PARATE, ALTO |
| POALL | ET, ESCTODO | OUTPUT | RESPUESTA, DEVUELVE |
| POTS | EP, ESCTS | RUN | CUMPLE, EJECUTA |
| PO | EI, ESCPROC | REPEAT | REPETIR, REPITA |
| POPS | ESCPROCS | TYPE | ESCRIBE |
| PONS | EN, ESCNS | PRINT | IMPRIME |
| RECYCLE | RECICLA | SHOW | MUESTRA |
| NODES | NODOS | SETCURSOR | PONCURSOR |
| BACK | ATRAS, RETROCEDE | RC | LEECARACTER, LEECAR |
| FORWARD | ADELANTE | RL | LEELINEA, LEELISTA |
| CS | BORRAPANTALLA | KEYP | LC?, TECLA? |
| CLEAN | BORRA, LIMPIA | BEEP | SONIDO |
| RIGHT | DERECHA, GIRADERECHA | NOBEEP | SILENCIO |
| LEFT | IZQUIERDA, GIRAIZQUIERDA | LOAD | RECUERDA, CARGA |
| HOME | CENTRO | CATALOG | CATALOGO |
| PX | PLUMAINVERSA, LAPIZREVES | SAVE | GUARDA |
| | | ERF | ELIMINAARCHIVO |
| PE | PLUMADEBORRAR, GOMA | .PRIMITIVES | .PRIMITIVAS |
| PD | CONPLUMA, CONLAPIZ | .EXAMINE | .EXAMINA |
| PU | SINPLUMA, SINLAPIZ | .DEPOSIT | .DEPOSITA |
| HT | OCULTATORTUGA | .CALL | .LLAMA |
| ST | MUESTRATORTUGA | | |
| SHOWNP | VISIBLE? | | |
| SETBG | COLORFONDO, PONCF | | |
| BG | FONDO, COLORFONDO | | |
| SETX | PX, PONX | | |
| SETY | PY, PONY | | |
| SETPOS | PXY, PONPOS | | |

Existen algunas versiones del lenguaje LOGO cuyo vocabulario de palabras reservadas se encuentra en castellano. Aunque los nombres no coinciden en todas las versiones, las más utilizadas son las que se exponen en la tabla. Las palabras clave se relacionan en el mismo orden en el que se han estudiado a lo largo de los capítulos de la obra dedicados al LOGO.

## Primitivas especiales

En LOGO se denomina primitiva a toda palabra clave o palabra reservada. Estas coinciden con los comandos y operadores propios del LOGO. Algunas primitivas tienen un carácter especial; éstas son las que van a constituir el objeto de los próximos párrafos.

Las primitivas especiales se caracterizan poque su nombre comienza por un punto. La más inmediata es PRIMITIVES cuya misión es la de mostrar todas las palabras reservadas. Este comando resulta muy útil para identificar las palabras claves propias de cada versión del LOGO. Un grupo destacable de primitivas especiales es el que acoge a las que afectan a las «interioridades» de la máquina. Dos de ellas se utilizan para acceder a posiciones de memoria. Por ejemplo, la función EXAMINE admite un dato de entrada numérico y entrega como salida el contenido de la posición de memoria apuntada por el referido dato de entrada. El comando opuesto es DEPOSIT que se nutre de dos datos de entrada. El primero determina la posición de memoria y el segundo el dato a almacenar en la misma.

Con estas primitivas y un ligero conocimiento del código máquina se pueden crear rutinas en dicho lenguaje. Para encadenar rutinas (creadas por éste u otro método) con procedimientos LOGO se dispone de la orden .CALL (llamar). Su efecto no es otro que el de ejecutar la rutina en código máquina situada a partir de la posición de memoria especificada. La posición se indica como dato numérico de entrada a la orden .CALL.

Como última primitiva cabe mencionar a .SETSCR (de SET SCReen, poner pantalla). Este comando define la relación entre las posiciones horizontales y verticales, lo que equivale a fijar la relación entre la resolución vertical y horizontal de la pantalla.

Utilizando .SETSCR de forma adecuada será posible ejecutar la representación en pantalla para evitar, por ejemplo, que los cuadrados aparezcan como rectángulos. El problema cuya resolución se encomienda a .SETSCR, deriva de las diferencias que surgen en la visualización de un mismo trazado por parte de distintos receptores de TV domésticos.

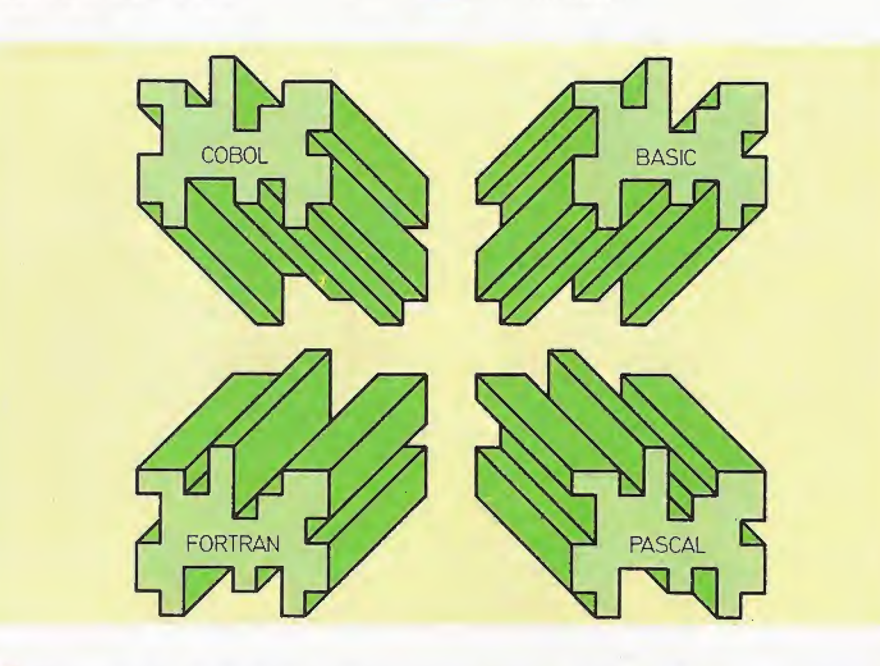
# Módula—2

## ¿El sucesor del Pascal?



El lenguaje Pascal, primero en utilizar plenamente los principios de la programación estructurada, ha servido de base a otros muchos, entre ellos el ADA, que pretendían potenciarlo y suprimir sus limitaciones. Consciente de este hecho, y con la experiencia acumulada tras varios años de trabajo, su creador, Nicklaus Wirth, presentó en 1980 una extensión de este popular lenguaje, que denominó Modula—2.

Modula—2 expande el Pascal en dos direcciones: hacia arriba, alejándose de la máquina, a través de la programación orientada a objetos; y hacia abajo, hacia la máquina, con un conjunto de funciones de bajo nivel (manipulación de bits, etc.) similares a las del lenguaje C.




*El mundo de la informática cuenta ya con decenas, e incluso tal vez centenares, de lenguajes de programación, bien dedicados o de propósito general. ¿Qué falta hace que se continúe investigando para el desarrollo de otros nuevos?*

## Un poco de historia

La primera pregunta que cabe hacerse acerca de este nuevo lenguaje es: ¿qué falta hace? En la actualidad existen bastantes decenas de lenguajes de programación, con distintas orientaciones, que pueden servir para realizar casi cualquier tarea que se proponga un programador. La respuesta es sencilla: en los últimos quince años, tanto los costes de desarrollo como la complejidad del software han crecido enormemente. Este hecho, unido a la gran fiabilidad que se exige en la actualidad a los programas de ordenador, ha originado lo que los científicos e ingenieros conocen como «la crisis del software».

No es raro que los grandes programas de ordenador se terminen fuera de plazo, presenten fallos de concepción o de diseño, o no cumplan totalmente las especificaciones pedidas. Paralelamente, el proceso de modificación o corrección es enormemente complejo, limitándose a «parchear» la zona problemática, en la esperanza de que el añadido no perturbe la función del resto del programa.

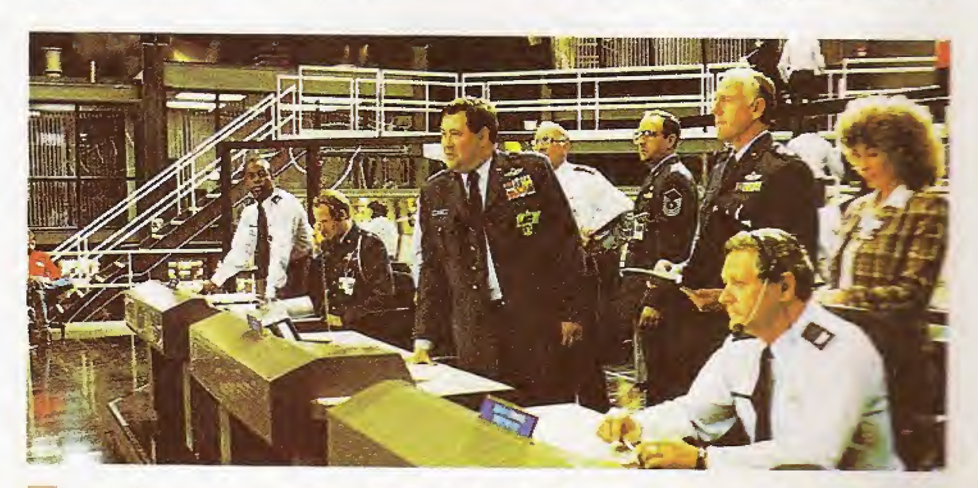
Los ingenieros de software han ido desarrollando, para resolver esta crisis, un conjunto de normas, entre las cuales destaca la necesidad de la sistematización del proceso creador como principal camino hacia la reducción de costes y el incremento de la fiabilidad.

Uno de los primeros científicos que se interesó por las circunstancias antes mencionadas fue Nicklaus Wirth, que dio a luz en 1970 el lenguaje Pascal, el primero en utilizar al máximo los principios de la programación estructurada. Pronto este lenguaje se convirtió en el favorito de los profesores, y en la actualidad se utiliza, entre otros lugares, en un gran número de universidades.

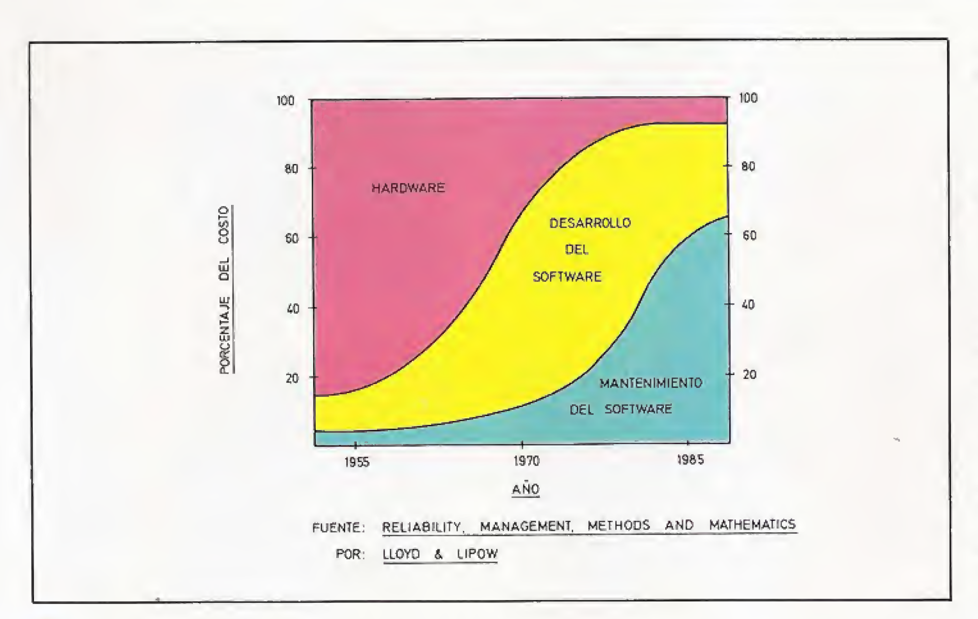
Sin embargo, Pascal no estaba creado para el desarrollo de software de gran complejidad, y pronto se vio la necesidad de un lenguaje que lo complementara en proyectos de gran altura.

En 1975, el Departamento de Defensa norteamericano juntó un nutrido grupo de científicos que, en 1983, puso en circulación el ADA, lenguaje que se convertiría en el estándar de dicho departamento.

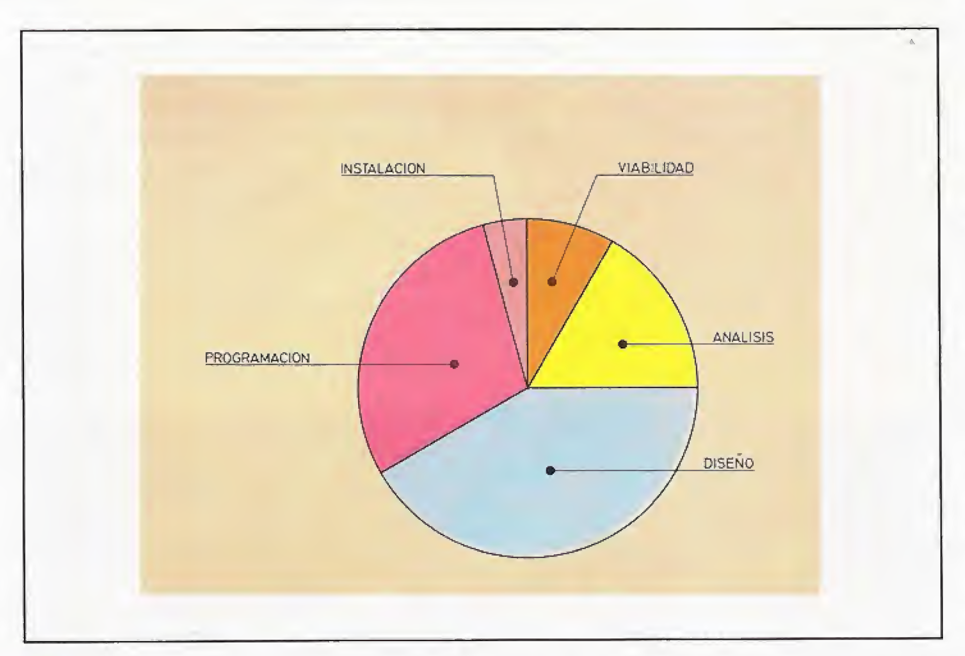
Nicklaus Wirth no estaba, mientras



*La aplicación de los ordenadores en proyectos de gran importancia (control de aviones, manejo de centrales nucleares, dirección de proyectos espaciales) hace precisos unos procesos de diseño que permitan asegurar la inexistencia de errores en los programas creados.*




*El elemento lógico —el software— ha ido incrementando su participación en el coste total de todo sistema informático. Diversos proyectos de investigación buscan reducir estos costes a la mínima cantidad posible.*




*La duración de cada una de las fases de elaboración de un proyecto informático es muy desigual. Como se muestra en la figura, las fases de diseño y programación son las más largas. La programación estructurada intenta reducir el tiempo empleado en cada uno de dichos procesos.*

tanto, ocioso. En 1980 introdujo el Modula—2, una extensión del Pascal preparada para el desarrollo de programas de gran complejidad, con muchas de las capacidades y características de paralelismo encontradas en ADA, y con la claridad, sencillez y compacidad del Pascal.

Desde entonces hasta nuestros días, el Modula—2 se ha ido extendiendo y ganando adeptos. En la actualidad se han desarrollado con él, o están en fase de desarrollo, muchos grandes proyectos de software.

## La estructura de Modula—2

Como la mayoría de los lenguajes de programación modernos, el Modula—2 es un lenguaje de formato libre, esto es, no posee restricciones en cuanto al número de espacios o posición de cada campo en la línea, como Fortran o Cobol.

La sintaxis de este lenguaje es muy similar a la del Pascal (no en vano ambos han sido creados por la misma persona), aunque hay algunas diferencias, menores pero importantes:

— Modula—2 distingue, en los identificadores, entre mayúsculas y minúsculas; esto es, las variables «MiVariable» y «mivariable» son distintas. Esta característica aumenta la claridad de los programas, aun a costa de incrementar ligeramente la dificultad de programación.

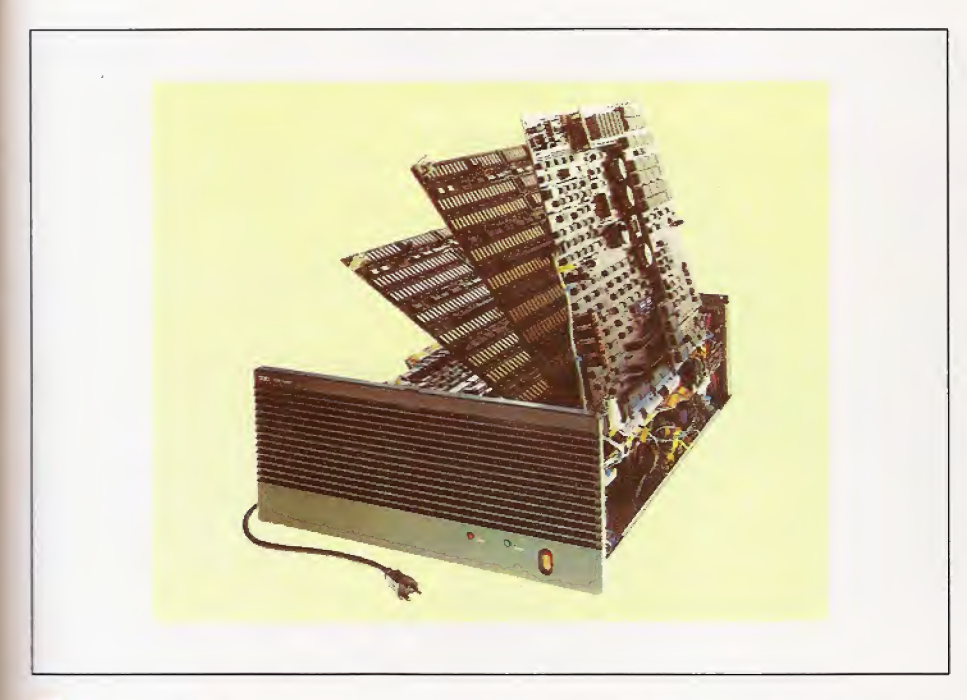
— Los comentarios pueden anidarse, lo que permite la eliminación temporal de zonas del programa sin más que situarlas entre dos delimitadores de comentario. Este hecho facilita enormemente la depuración del software.

## Algunas diferencias con Pascal

En un programa en Modula—2 pueden declararse cinco entidades distintas: constantes, tipos, variables, procedimientos y funciones. Los conocedores de Pascal echarán de menos las etiquetas (labels), suprimidas en este lenguaje, ya que no está permitida la ruptura del flujo.

Siguiendo con las diferencias con Pascal, un programa en Modula—2 puede contener cualquier número de declaraciones de tipos, constantes o variables, en cualquier orden. De esta forma puede definirse cada identificador cerca de la zona en la que vaya a ser usado, lo que aumenta la claridad del programa.

Por lo que se refiere a las declaraciones de constantes, ahora se admite la definición de «expresiones constantes»



*La independencia de la máquina es otro de los fines que se han propuesto los investigadores en software. Modula-2 consigue esta independencia a través de módulos especializados en la interacción con la arquitectura subyacente.*

que relacionan entre sí dos o más valores ya definidos. Esto permite modificar un conjunto de constantes interrelacionadas sin más que modificar una de ellas.

## Tipos de datos

Un «tipo de datos» puede definirse como un par <V, O>, donde

V es un conjunto de valores, y

O es un conjunto de operaciones definidas sobre esos valores.

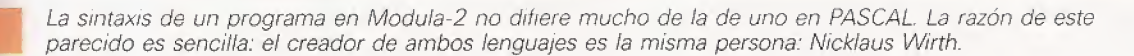
Modula—2 incluye un conjunto de tipos predefinidos y permite la definición de otros nuevos por parte del usuario. Nos detendremos ahora brevemente en los tipos similares a los del Pascal, dejando para más tarde los tipos abstractos de datos, uno de los puntos fuertes del nuevo lenguaje.

Existen seis tipos de datos escalares: enteros (INTEGER) y cardinales (CARDINA), reales (REAL), booleanos (BOOLEAN), caracteres y cadenas de caracteres (CHAR y ARRAY OF CHAR, respectivamente), enumerativos y subrangos. Salvada la introducción del tipo cardinal (enteros positivos), las diferencias con Pascal son mínimas, y escapan del propósito de presentación general de la obra, por lo que no nos detendremos en ellas.

En cuanto a los tipos estructurados de datos (tipos compuestos por otros más primitivos), cabe clasificarlos en función de tres categorías: tamaño (determinado, como en las matrices, o indeterminado: ficheros), homogeneidad (matrices frente a registros, por ejemplo) y métodos de acceso (directo o secuencial). Una revisión rápida permite encontrar en este lenguaje matrices (ARRAY), registros (RECORD) y conjuntos (SET).

```
MODULE TriangularNumbers;
(* Este programa calcula los primeros 25 números triangulares *)
FROM InOut IMPORT WriteCard, WriteLn;
VAR
  n   : CARDINAL;
  tri : CARDINAL;

BEGIN
  tri:= 0;
  FOR n:= 1 TO 25 DO
    tri:= tri + n;
    WriteCard (tri, 4);
    WriteLn;
  END
END TriangularNumbers;
```



*La sintaxis de un programa en Modula-2 no difiere mucho de la de uno en PASCAL. La razón de este parecido es sencilla: el creador de ambos lenguajes es la misma persona: Nicklaus Wirth.*




*En la actualidad existen compiladores de Modula-2 para distintos tipos de ordenadores, personales, minis o mainframes. La portabilidad entre unos y otros sistemas es uno de los principales aspectos buscados por su creador.*

Al contrario que Pascal, Modula—2 no tiene un tipo «fichero» predeterminado, sino que ofrece un módulo de biblioteca (más tarde veremos qué es esto) que permite la utilización de este tipo de estructuras.

Como tipo de datos dinámico, Modula—2 suministra, al igual que Pascal, el «puntero» (POINTER); el tratamiento en uno y otro lenguajes es muy similar.

## Conversión entre tipos de datos

La sintaxis de Modula—2 es, por lo que respecta a lo visto hasta ahora, similar a la del Pascal, con algunas pequeñas, aunque importantes diferencias. Entre éstas se encuentra la provisión por parte del primero de un conjunto de operadores para la conversión y transferencia entre entidades de distinto tipo. Por conversión se entiende el pasar un valor de un tipo a otro, manteniendo el valor conceptual y variando la representación interna de la máquina (por ejemplo, al pasar un entero a real se varía la forma de almacenamiento, pero se mantiene la categoría «número»), mientras que la transferencia implica el mantenimiento de la representación interna y el cambio del valor conceptual (como sucede al hacer una transformación entre entero y carácter).

Para la conversión de tipos, Modula—2 ofrece las funciones TRUNC (truncar; permite la conversión de un número real a cardinal) y FLOAT (para la conversión de un cardinal a real). Por lo que se refiere a la transferencia de tipos, están disponibles las funciones CHAR (transferencia de un cardinal a carácter), ORD (conversión en cardinal de cualquiera de los siguientes tipos: carácter, booleano, entero, enumerativo o cardinal), y VAL (transfiere al tipo indicado en primer lugar el valor que señale el segundo parámetro).

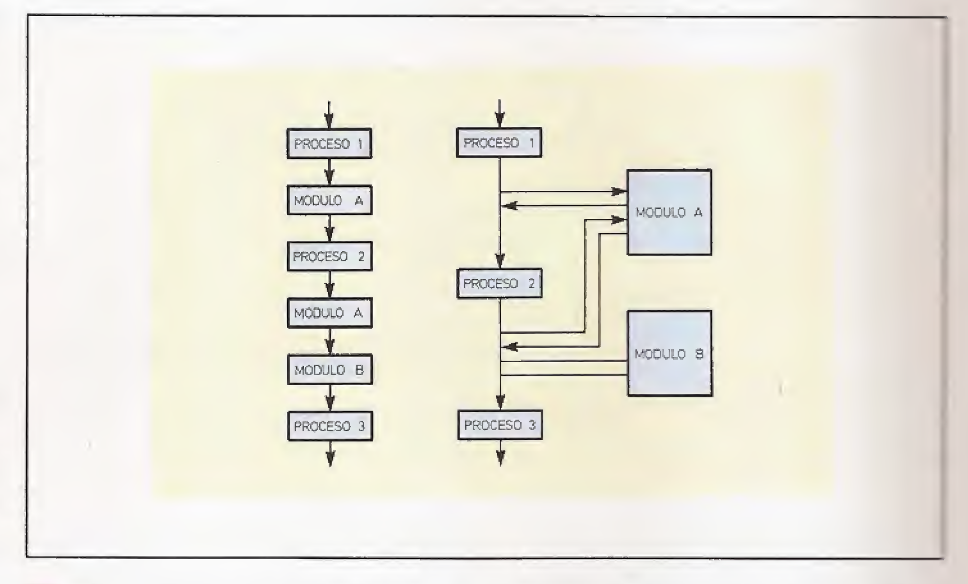
Existe una segunda forma de realizar transferencias entre tipos que no tienen paralelo en Pascal: cualquier identificador de tipo puede usarse como un identificador de función para transferir un valor de un tipo a otro. La única restricción a esta regla es que los dos tipos deben tener la misma representación interna. Así, por ejemplo, una operación del tipo

entero + cardinal,

no permitida, puede realizarse de las dos formas siguientes:

entero + INTEGER(cardinal), con lo que el resultado es entero, o

CARDINAL(entero) + cardinal, que da como resultado un cardinal.




*Un programa modular puede verse de dos formas diferentes: como una sucesión encadenada de módulos que se ejecutan uno tras otro, o como un conjunto de llamadas a subrutinas cerradas y contenidas en cada uno de los módulos.*

## Estructuras de control

Las estructuras de control de cualquier lenguaje de alto nivel pueden dividirse en tres grupos principales: secuencial (ausencia de estructuras de control de flujo), alternativo (selección entre dos o más posibilidades), y recursivo/iterativo (programación de bucles). Modula—2 ofrece además una instrucción de control de flujo muy especial

```
MODULE CharacterCounter;

(* Este programa cuenta los caracteres de una línea de entrada *)

FROM InOut IMPORT

   (* const *) EOL,
   (* proc  *) Read, WriteString, WriteCard, WriteLn;

VAR count: CARDINAL;
    ch   : CHAR;

BEGIN
   count:= 0;
   LOOP
     Read(ch);
     IF ch = EOL THEN EXIT END;
     IF ch <> ' '
       THEN INC (count)
     END
   END;
   WriteString('Número de caracteres: ');
   WriteCard(count, 5);
   WriteLn
END CharacterCounter.
```

*Aspecto de un módulo de programa, característica denotada por el identificador MODULE situado al comienzo. Pueden observarse las zonas de importación de estructuras, la construcción LOOP/EXIT/END, y el nombre del módulo repetido al final de éste.*

```
DEFINITION MODULE ArrayMaxAndMin;

EXPORT QUALIFIED
  (* type *) RealArray,
  (* proc *) MaxPos, MinPos;

  TYPE RealArray = ARRAY[1..1000] OF REAL;

  PROCEDURE MaxPos (a: RealArray) : CARDINAL;
  PROCEDURE MinPos (a: RealArray) : CARDINAL;

END ArrayMaxAndMin;
```

*Un módulo de definición se caracteriza, entre otras cosas, por la palabra DEFINITION antepuesta a MODULE. En él se especifican únicamente los tipos de datos, las variables, y las cabeceras de los procedimientos. No contiene código.*

HALT, que detiene completamente el programa y devuelve el control al sistema operativo. Esta instrucción puede ser muy útil al detectarse errores que impidan la continuación y terminación normal del programa. Debe emplearse con infinita precaución y, por supuesto, sin abusar de ella para resolver situaciones en las que pueda utilizarse cualquier otra forma de control de flujo.

La estructura alternativa por excelencia, IF THEN ELSE END, presenta respecto de su homóloga Pascal dos diferencias sustanciales: la presencia obligatoria de la cláusula END final, que permite evitar la ambigüedad que de otro modo surge en la asignación de un ELSE en una anidación de IF's, y la posibilidad de simplificar los saltos multiopción mediante la unificación en una sola palabra clave, ELSIF, de las cláusulas ELSE e IF. La otra estructura alternativa, CASE OF, difiere de su homónima en tres aspectos esenciales: la separación entre campos, que utiliza una barra vertical, la posibilidad de incluir subrangos como etiquetas de selección, y la presencia de la cláusula final ELSE, para contemplar los valores de la variable de control no especificados como etiquetas.

Las estructuras iterativas no difieren significativamente de las ya conocidas por un usuario de Pascal, salvo en el caso de WHILE DO END y FOR TO DO END, donde se ha incluido el END final para suprimir la necesidad de utilizar sentencias compuestas. En esta última estructura, FOR, puede introducirse ahora una cláusula opcional, BY, que indique el incremento de la variable de control.

Se ha añadido además en Modula—2 una estructura que, aunque pueda parecer absurda, es de gran utilidad: el bucle infinito, delimitado por LOOP y END. Para poder salir de él se emplea la cláusula EXIT.

## Procedimientos y funciones

Modula—2 admite tres clases de subprogramas: procedimientos, funciones, y módulos. Los dos primeros son similares a sus correspondientes en Pascal, por lo que nos limitaremos a señalar las diferencias más relevantes, mientras que los terceros, completamente nuevos, serán discutidos más adelante.

Las principales diferencias entre los procedimientos y funciones de Pascal y Modula—2 aparecen en la sintaxis de uno y otro lenguaje. Así, en el segundo es obligatoria la lista de parámetros en la cabecera, aunque esté vacía; es necesario así mismo indicar el nombre del procedimiento tanto al principio (en su cabecera) como al final, lo que facilita la identificación de los distintos END's que suelen acumularse en este lugar.

En Modula—2 la palabra reservada FUNCTION es sustituida por PROCEDURE. La distinción entre procedimientos y funciones se realiza según el tipo de cabecera: en la de estas últimas aparece el tipo de dato que devuelven.

Otra variación significativa en lo que a funciones se refiere es la forma en que se devuelven los resultados: a través de la cláusula RETURN, en lugar de mediante su asignación al nombre de la función.

## Un programa en Modula—2

La estructura de un programa en Modula—2 no difiere grandemente de la de un programa en Pascal, aunque, como veremos, tampoco se asemeja a ella como una gota de agua a otra.

Un programa en Modula—2 está dividido en módulos, que pueden ser de cuatro tipos distintos: de programa, internos, de definición, y de implementación. El clásico programa Pascal se corresponde, con algunas diferencias, con un módulo de programa. Veamos esas «algunas diferencias»:

— En Modula—2, la palabra reservada PROGRAM se sustituye por MODULE.
— Un módulo de programa no emplea parámetros.
— Al final del módulo debe consignarse obligatoriamente su nombre.

Por lo que se refiere a la sintaxis, cabe señalar que el siempre presente punto y coma se emplea ahora como separador de sentencias, no como terminador (de forma similar al lenguaje C. En aquel entonces se explicó ya la diferencia entre una y otra concepciones, por lo que no vamos a entrar ahora de nuevo en ello).

## Procedimientos de entrada/salida

Las facilidades de entrada/salida de Modula—2 no son implícitas al lenguaje, sino que vienen incluidas en un módulo especial denominado InOut, lo que permite eliminar la dependencia entre software y hardware: basta variar dicho módulo, en lugar de todo el compilador, si se desea ejecutar los programas creados sobre otra arquitectura.

Los procedimientos estándar ofrecidos por InOut son Read (lee un carácter del dispositivo de entrada, cualquiera que éste sea), Write (escribe un carácter en el dispositivo de salida), ReadString y WriteString (lee o escribe, respectivamente, una cadena de caracteres), ReadCard y WriteCard (para la lectura y escritura de cardinales), ReadOct y WriteOct (para la lectura y escritura de números en octal), y otros varios, adecuados a los distintos tipos de datos. Están asimismo disponibles los procedimientos ReadLn y WriteLn, para el cambio de línea, Done, para indicar el final de la entrada (similar al EOF de Pascal), y OpenInput, OpenOutput, CloseInput y CloseOutput para la redirección de la entrada y/o la salida.

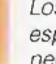
Para el trabajo con ficheros se dispone del módulo FileSystem, que provee, aparte de la definición del tipo FILE (fichero), algunas herramientas de bajo nivel para su tratamiento.

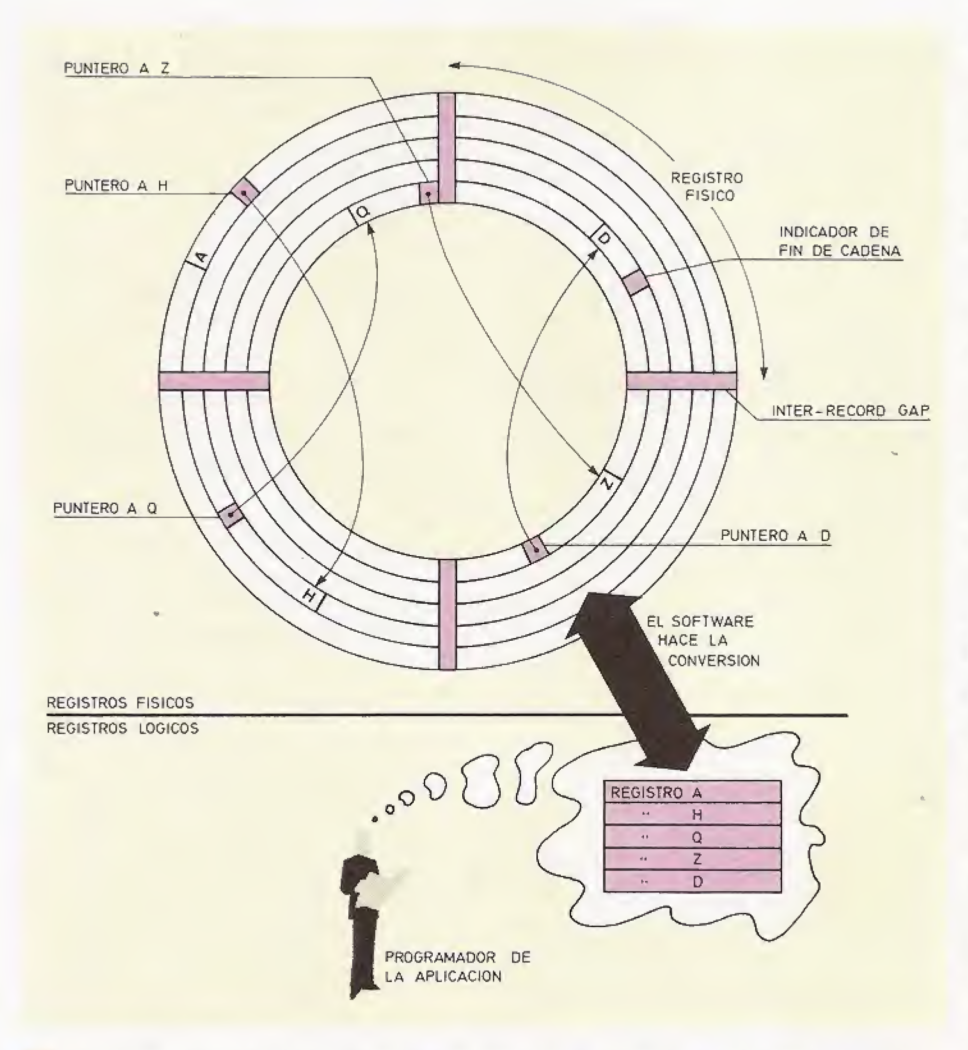
## La biblioteca «estándar» de Modula—2

La reciente creación de Modula—2 es uno de los principales factores que han impedido su estandarización, por lo que no puede hablarse en puridad de nada «estándar» en él. No obstante, la mayoría de las implementaciones se basan en la definición de Wirth, por lo que acudiremos a ella cuando sea necesario hablar de características «normalizadas».

Entre los módulos presentes en todo compilador de Modula—2 deben hallar-

```
IMPLEMENTATION MODULE ArrayMaxMin;
  PROCEDURE MaxPos (a: RealArray) : CARDINAL;
  VAR
    (* Variables *)
  BEGIN
  (* Aquí vendría el cuerpo del procedimiento *)
  END MaxPos;
  PROCEDURE MinPos (a: RealArray) : CARDINAL;
  VAR
    (* Variables *)
  BEGIN
  (* Aquí vendría el cuerpo del procedimiento *)
  END MinPos;
END ArrayMaxAndMin.
```


*Los módulos de implementación deben contener el código preciso para efectuar las distintas tareas especificadas en el módulo de definición. Pueden contener así mismo el código de inicialización necesario para dar valores a las distintas variables la primera vez que se llame al módulo.*




*El almacenamiento real o físico de los datos en la memoria del ordenador no se corresponde con su organización lógica. Para independizar el programa, en lo posible, de la máquina, se emplean módulos de enlace, como el módulo SYSTEM.*

se una biblioteca de funciones matemáticas (MathLib, que incluye funciones como la raíz cuadrada, valor absoluto, logaritmo neperiano, etc.), otra para el control del almacenamiento en memoria (Storage, con procedimientos para la gestión de punteros), y algunas otras como Arithmetic (operaciones con números: máximo, mínimo, etc.), Utilities (utilidades varias), etc.

## Módulos: qué y cuáles son

El módulo es, aparte de la característica que da nombre al lenguaje, la más importante innovación de Modula—2. Como ya se ha indicado, existen cuatro tipos diferentes de módulos: de programa, internos, de definición y de implementación; estos dos últimos suelen englobarse en el nombre genérico de módulos «de biblioteca» (library modules).

Los módulos de programa son programas completos, y pueden importar recursos software de otros módulos. Se han comentado ya en un apartado anterior, por lo que no volveremos sobre ellos.

Los módulos de biblioteca están divididos en dos partes, que pueden ser compiladas separadamente: especificación y cuerpo, que se corresponden, respectivamente, con los módulos de definición e implementación.

Los módulos internos, por fin, son estructuras que pueden ser definidas dentro de cualquier programa para facilitar su inteligibilidad o limitar el campo de aplicación de ciertas estructuras o variables.

La separación del programa en módulos permite la compilación separada: los distintos módulos pueden compilarse y guardarse separadamente, y al final únicamente es necesario montar juntos los diferentes módulos que forman un programa para obtener éste. De esta forma cualquier módulo creado se incorpora a la biblioteca de recursos de que dispone el programador, lo que evita la repetición de estructuras semejantes en distintos programas.

Para la interrelación de módulos se emplean dos cláusulas especiales: IMPORT, que permite especificar de dónde y qué se necesita en el módulo actual (por ejemplo, para poder escribir cadenas de caracteres necesitamos la función WriteString, que se encuentra en el módulo InOut). La sentencia que permite utilizarla es:

```
FROM InOut IMPORT WriteString;
```

y EXPORT, que permite definir las estructuras de un módulo que podrán ser importadas por otro (así, si hemos creado el procedimiento Alarma y deseamos que otros programadores puedan utilizarlo, utilizaremos en el módulo que lo contenga la sentencia EXPORT Alarma). La exportación puede ser cualificada (para utilizar el procedimiento es necesario mencionar el nombre del módulo en el que se encuentra) o no.

## Módulos de biblioteca

Los módulos de biblioteca son un conjunto de módulos de los que el programador dispone para introducirlos en su programa. Provienen de dos fuentes principales: unos pertenecen al sistema de desarrollo en Modula—2, mientras que otros han sido creados con anterioridad por el propio programador.

Un módulo de biblioteca consta de dos zonas bien diferenciadas: la zona de definición (DEFINITION MODULE), en la que se especifican las distintas caracte-

rísticas del módulo, los procedimientos que contiene, y las estructuras que puede importar y exportar, y otra de implementación (IMPLEMENTATION MODULE), que contiene el cuerpo de los procedimientos mencionados en el módulo de definición. De esta forma, si se desea variar la implementación de un módulo no debe recompilarse todo el programa, sino sólo ese módulo. Los módulos de definición son normalmente públicos, de forma que un programador pueda conocer de qué estructuras dispone dentro de él.

Los módulos de implementación se distinguen de los módulos de programa en la palabra IMPLEMENTATION que antecede a MODULE. Por lo demás son exactamente iguales. Una facilidad añadida a los módulos de implementación es que pueden tener código de inicialización, que se ejecutará la primera vez que se llame al módulo, de forma que las variables, por ejemplo, partan de valores conocidos.

Los módulos de biblioteca permiten crear estructuras abstractas, similares a cajas negras que ofrecen determinados servicios, sin importar cómo se implementan realmente esos servicios. De esta forma se facilita la creación de software y la interacción entre un número elevado de programadores.

Los módulos internos van introducidos indistintamente en un módulo de programa o en uno de implementación. Su función es aislar estructuras de datos, procedimientos, variables o cualesquiera otras estructuras del resto del módulo que los contenga, con el fin de limitar el campo de visibilidad o para mejorar la legibilidad del conjunto.

## Abstracción de datos

El proceso de abstracción puede definirse como «identificar los conceptos esenciales e ignorar los detalles particulares». Este proceso es una de las mayores herramientas conceptuales de que dispone un usuario para enfrentarse al problema de la complejidad del software, por lo que nos detendremos brevemente en él.

Los tipos predefinidos en Modula—2 pueden considerarse como abstracciones de datos de bajo nivel. En efecto, permiten al programador no tener en cuenta aspectos tales como la representación interna de los datos o los componentes hardware que permiten su almacenamiento y tratamiento.

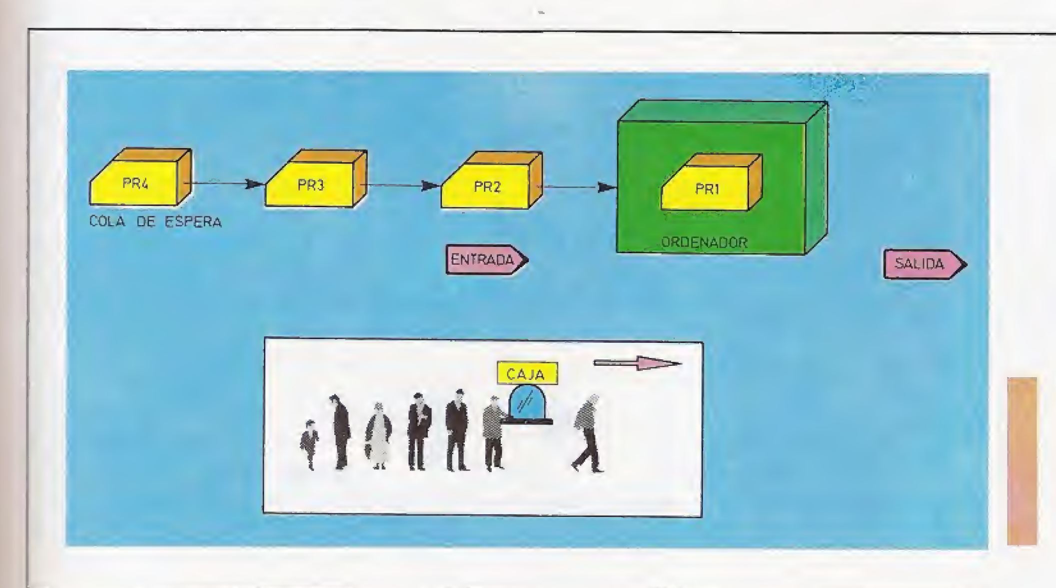
Entre las ventajas de la abstracción están la posibilidad de concentrarse en el problema, sin necesidad de recordar detalles de implementación que no son necesarios en ese momento determinado. Por otra parte, esta situación permite asegurar la integridad de los datos, puesto que el usuario no puede modificar la estructura interna más que a través de las facilidades que se le otorguen para ello. La tercera ventaja de la abstracción es la posibilidad de cambiar la implementación de la estructura oculta sin necesidad de variar el programa en consecuencia; únicamente es necesario atenerse a la definición de interfaz existente. Y cuarto y último, la abstracción de datos permite unir en una sola entidad todos los aspectos relacionados con ella, con lo que esto representa en claridad y legibilidad.

## Abstracciones de bajo nivel

Para desempeñar su trabajo, muchas veces los programadores necesitan conocer y utilizar detalles propios de la arquitectura sobre la que estén desarrollando software: almacenamiento en memoria, manipulación de direcciones físicas, etc. Aunque el lenguaje Modula-2 ha sido desarrollado para ser independiente de la máquina sobre la que esté siendo ejecutado, existe un módulo que permite utilizar estas características de bajo nivel: el módulo SYSTEM (sistema). Su utilización debe hacerse con el máximo cuidado, ya que cualquiera de los procedimientos o los tipos por él exportados pueden ocasionar la pérdida de compatibilidad del programa.

El módulo SYSTEM ofrece al programador distintos tipos y procedimientos, entre los que se encuentran:

— El tipo WORD, que se correspon-




*Normalmente la ejecución de un programa es secuencial: las subrutinas, módulos, o como quiera llamárseles, se ejecutan una tras otra, sin tener en cuenta el proceso anterior o posterior.*




*La capacidad de proceso de los ordenadores actuales permite la generación de programas concurrentes, en los que distintas rutinas se ejecutan de forma simultánea. Modula-2 provee de diversas herramientas para el desarrollo de programas de este tipo.*

de exactamente con una palabra de memoria.

— El tipo BYTE, cuya representación en memoria es un byte físico.

— Los tipos HALFWORD (media palabra) y LONGWORD (doble palabra).

Estos tipos son incompatibles con cualesquiera otros admitidos por el Modula—2 o creados por el usuario. La única operación que puede efectuarse con ellos es la asignación. Puesto que algunas arquitecturas pueden no basar su distribución de memoria en palabras y bytes, algunos compiladores carecen de ellos.

Como dato interesante y muy útil podemos indicar que si en algún procedimiento se pasa un parámetro definido como ARRAY OF WORD o ARRAY OF BYTE puede asignársele cualquier variable actual, independientemente de su tamaño y tipo. Esta peculiaridad hace a estos tipos extremadamente útiles a la hora de desarrollar estructuras genéricas.

— El tipo ADDRESS (dirección), que puede entenderse como un POINTER TO WORD.

— Y, por último, los procedimientos ADDR (devuelve la dirección en memoria de la variable que se le pase como parámetro), SIZE (devuelve el tamaño de la variable que se le pase como parámetro) y TSIZE (devuelve el tamaño de un dato de tipo determinado). Estos procedimientos se utilizan preferentemente para conocer la cantidad de memoria asignada a una estructura de datos determinada.

| Palabras reservadas de Modula—2 | |
|---|---|
| AND | LOOP |
| ARRAY | MOD |
| BEGIN | MODULE |
| BY | NOT |
| CASE | OF |
| CONST | OR |
| DEFINITION | POINTER |
| DIV | PROCEDURE |
| DO | QUALIFIED |
| ELSE | RECORD |
| ELSIF | REPEAT |
| END | RETURN |
| EXIT | SET |
| EXPORT | THEN |
| FOR | TO |
| FROM | TYPE |
| IF | UNTIL |
| IMPLEMENTATION | VAR |
| IMPORT | WHILE |
| IN | WITH |

## Trabajando con procesos

La definición de proceso que cuadra en este apartado es «programa en ejecución». Un programa es una estructura estática, muerta; un proceso es una estructura dinámica, en constante evolución para producir unos resultados que dependen de cómo se ha desarrollado el programa que le ha dado vida.

A menudo es interesante ver un programa como un conjunto de procesos que se desarrollan simultáneamente. Esto se conoce como «programación concurrente». Modula—2 ofrece una serie de posibilidades para lanzar procesos desde uno determinado, facilitar la interacción entre los procesos padre e hijo y, tras la finalización del proceso hijo, borrarlo de la memoria del ordenador.

Las primitivas para el control de procesos dependen de la máquina sobre la que se esté trabajando, por lo que se alojan en el módulo SYSTEM, ya comentado con anterioridad. Entre ellas se encuentran el tipo PROCESS (proceso) y los procedimientos NEWPROCESS y TRANSFER (proceso nuevo y transferir, respectivamente).

En Modula—2, un proceso es un procedimiento sin parámetros, declarado en el nivel más alto (no contenido en ningún otro procedimiento). Un proceso puede tener asociado un conjunto de sentencias (el procedimiento sin parámetros) y un área de datos. Una vez que se ha hecho la asignación, es posible iniciar la ejecución del proceso. Pueden declararse y lanzarse cuantos procesos sean necesarios, y todos se ejecutarán simultáneamente.

Para ejecutar un proceso, o pasarle control desde otro, se emplea el procedimiento TRANSFER, que recibe como parámetros los nombres del proceso que tiene el control y del que lo recibe.

La sincronización de procesos puede hacerse a través de semáforos, buzones, áreas compartidas de memoria, o cualesquiera otros procedimientos que el programador crea convenientes. La profundidad del tema impide que sea tratado en esta obra de carácter introductorio.

# PASCAL (1)

## Las ventajas de la programación estructurada



l PASCAL es un lenguaje de programación de alto nivel, modular y estructurado. Ha de utilizarse siguiendo una serie de reglas muy precisas, que se detallarán en este primero y en sucesivos capítulos. Así como todos los lenguajes humanos incorporan un conjunto de reglas gramaticales, también existen reglas a las que hay que ajustarse para poder dialogar con la máquina en lenguaje PASCAL.

Para que un programa sea fácilmente comprensible y susceptible de ser modificado con posterioridad, debe ser claro, inteligible y corto. Estos requisitos son fáciles de conseguir en un programa escrito en PASCAL, debido a las propias características de este lenguaje. En PASCAL, los programas se escriben de forma secuencial y ordenada. El empleo de bifurcaciones incondicionales (instituciones que rompen la secuencia de ejecución del programa) es desaconsejable, ya que dificultan el análisis y seguimiento del mismo. Al evitar su presencia —lo cual es siempre posible—, se logrará una ejecución ordenada de las instrucciones. Por lo demás, un programa en PASCAL se puede dividir en pequeños trozos o módulos que otorgan una gran flexibilidad al programa en orden a futuras modificaciones.




*El PASCAL es un lenguaje informático de alto nivel, modular y estructurado.*

Otra característica básica del PASCAL es la de ser un lenguaje estructurado. Del mismo modo que a partir de los tres colores primarios —rojo, verde y azul—, mezclándolos adecuadamente, es posible conseguir toda la gama del espectro; así también, partiendo de tres simples estructuras, a saber: secuencial, de bifurcación y repetitiva, se puede escribir cualquier programa.

El concepto de estructuración se refiere tanto al programa en sí, como a los datos del mismo. Estos últimos pueden estructurarse para reducir la complejidad de los algoritmos de trabajo, aumentar la claridad del programa y, en definitiva, conseguir un mayor rendimiento.

El programa en PASCAL siempre debe ser compilado antes de proceder a su ejecución. Ello significa que se traduce de una sola vez a lenguaje máquina y no instrucción a instrucción, a medida que las va procesando.

El formato de escritura de un programa es libre, de tal forma que las sentencias pueden colocarse en cualquier parte del listado. La única restricción a esta libertad es que cada instrucción debe terminar con un punto y coma para diferenciarla de la siguiente.

Los espacios en blanco se pueden insertar libremente, con lo que el listado del programa gana en claridad y legibilidad, al poder utilizar márgenes y variables para cada módulo o porción del programa. En terminología informática, este método se llama *indentación* o profundidad lógica.

Las instrucciones se escriben en una notación similar al inglés restringido y al álgebra. Para ello se dispone de los siguientes caracteres reconocibles por el ordenador (letras en mayúsculas o minúsculas):

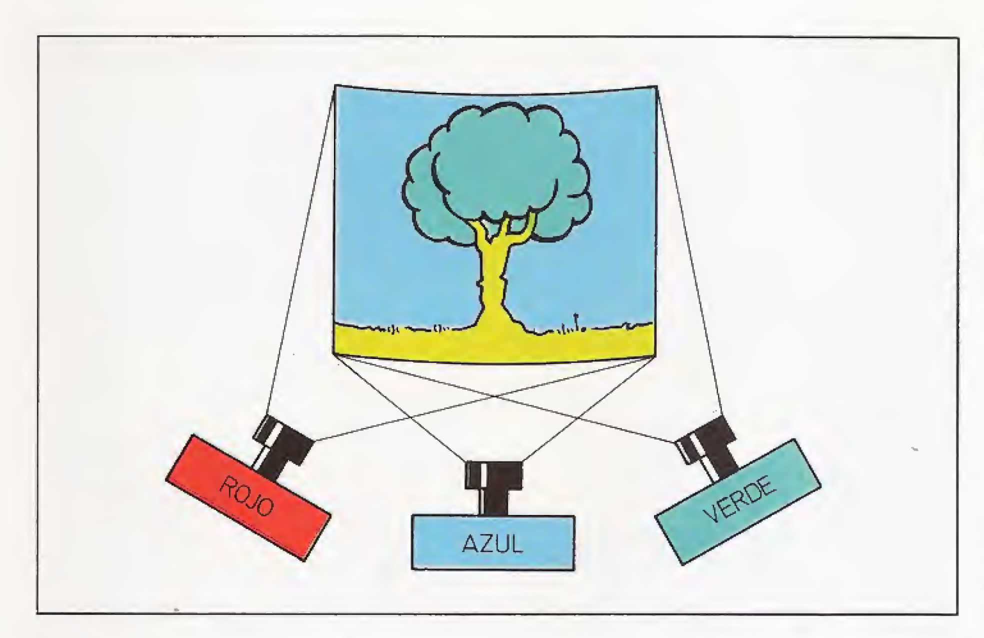
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 + - * / = : . , ( ) ' < > ; [ ]

Hay que destacar la distinción clara y precisa que existe entre, por ejemplo, los paréntesis y los corchetes; cada carácter tiene sus aplicaciones concretas y en ellas no puede ser sustituido por otro carácter similar.

Para incluir en el listado del programa los comentarios pertinentes, es preciso escribir el texto en cuestión encerrado entre paréntesis y asteriscos: "(*"y"*)". (*A que así es más fácil*). Desde luego, los comentarios pueden emplazarse en cualquier parte del programa.

## Identificadores

Cuando nacemos, nuestros padres nos conceden el privilegio de otorgarnos




Al igual que ocurre con las composiciones cromáticas, partiendo de tres únicas estructuras (secuencial, de bifurcación e iterativa) pueden construirse programas PASCAL adecuados para resolver las más diversas aplicaciones.

un nombre, por el que se nos puede identificar el resto de nuestra vida. De forma similar, los datos y elementos de un programa en PASCAL pueden recibir un nombre o *identificador,* por el que serán reconocidos en lo sucesivo dentro del programa.

El identificador proporciona, además, una idea del tipo de dato de que se trata y de su aplicación. Por ejemplo, si el dato coincide con la cantidad de vino producida en una determinada cosecha, puede otorgarse a este dato el nombre: VINO. Estos identificadores deben cumplir ciertas reglas; por ejemplo, deben empezar por una letra, aunque sus restantes caracteres pueden ser letras o cifras decimales. Su longitud no está limitada, si bien, el compilador sólo reconocerá los ochos primeros caracteres. Por ejemplo el identificador HIPOPOTAMOS, bajo la perspectiva del compilador, coincide con HIPOPOTAMAS.

Por otro lado, existen palabras con significado propio para el compilador que no se pueden utilizar como identificadores. Estas palabras reservadas reciben el nombre de *delimitadores* y tienen reservada una misión específica; por ejemplo, PROGRAM, BEGIN, IF, END...

## Aspecto de un programa en PASCAL

Un algoritmo (método de resolver un problema determinado) escrito en PASCAL, consta de dos zonas fundamentales:

- La descripción de las acciones a realizar, coincidente con las sentencias del programa.
- La descripción de los datos a utilizar, que son las declaraciones y las definiciones de parámetros.

A su vez, un programa en PASCAL está dividido en tres bloques independientes:

— Cabecera.
— Declaraciones y definiciones.
— El cuerpo de sentencias

La *cabecera* sirve para identificar al programa. Consta de la palabra reservada PROGRAM, seguida por un identificador válido PASCAL. Este último coincidirá con el nombre genérico que se le adjudique al programa.

A continuación, y encerrados entre paréntesis, figurarán los nombres de los ficheros o dispositivos lógicos a través de los que el programa se comunicará con el exterior. Estos ficheros son propios de cada sistema. Generalmente se llama INPUT al fichero de entrada (teclado) y OUTPUT al fichero de salida (impresora o pantalla).

La zona de *declaraciones y definiciones* es opcional y comprende las siguientes partes que, en el caso de que existan todas ellas, deben seguir el orden riguroso en el que se relacionan a continuación:

1. Declaraciones de etiquetas.
2. Definición de constantes.
3. Definición de tipos.
4. Declaración de variables.
5. Declaración de procedimientos y funciones.

Cada una de estas cinco partes se definirá más tarde con todo detalle.




Un programa en lenguaje PASCAL consta de tres bloques principales, que aparecen dentro del mismo en un riguroso orden.

El cuerpo de sentencias incluye a las sentencias ejecutables del programa; sentencias que quedarán encerradas por las palabras reservadas BEGIN y END. Tras el END aparecerá un punto final ineludible.

## Los datos del PASCAL

Los datos son la materia prima con la que el ordenador va a elaborar su producto, que no es otro que el resultado de la ejecución del programa. Los datos pueden aparecer en forma de valores *constantes* o representados por medio de *variables;* estos últimos pueden ver alterado su valor durante la ejecución del programa. Una semana tiene siete días, éste es un dato constante; mientras que el dinero de nuestro bolsillo suele ser un dato «bastante» variable.

Los datos se tienen que declarar en la zona de declaraciones y definiciones del programa, para que sea factible su uso posterior. Para declarar una constante se utiliza la siguiente notación:

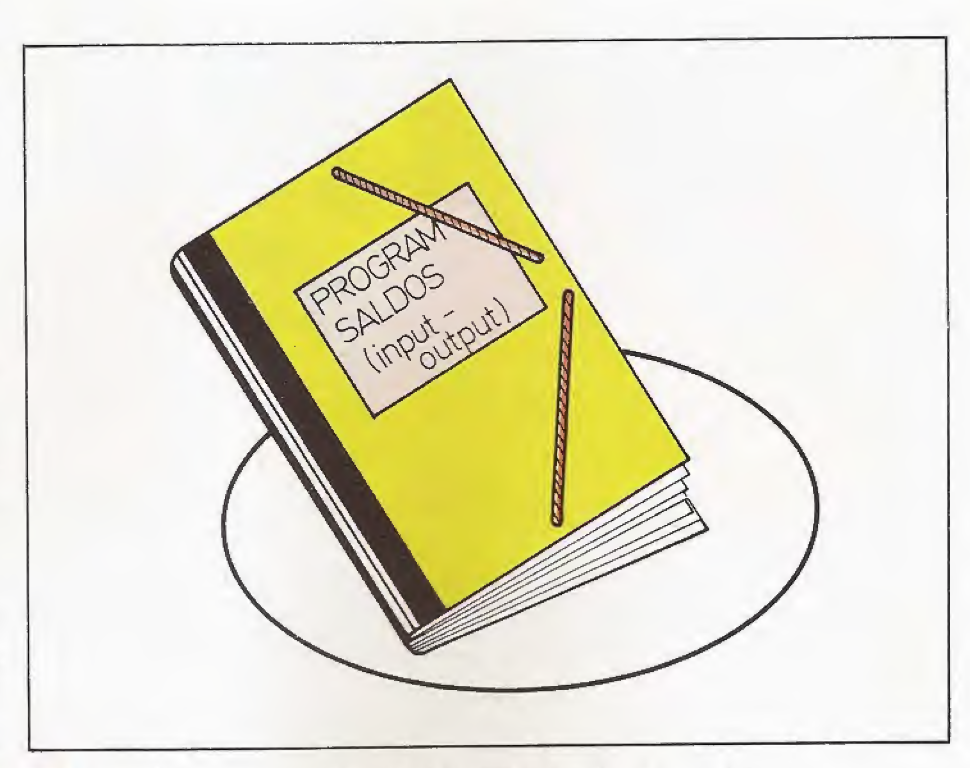
```
CONST <identificador> = <constante>;
```

Análogamente, las variables se declaran con la notación que sigue:

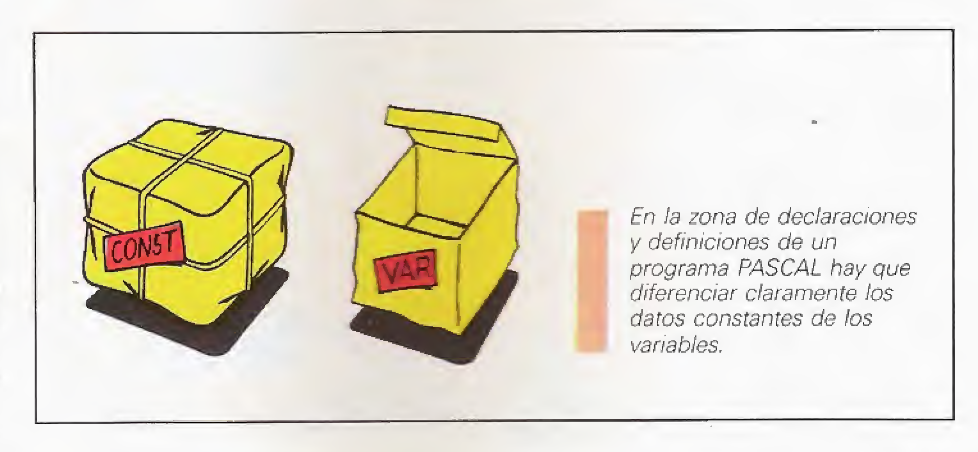
```
VAR <identificador1> [ <identificador2> , ...]: <tipo>;
```

El argumento *tipo* se refiere a la clase de variable de que se trata y sirve para indicar el conjunto de valores que la variable puede asumir. Por lo tanto, la declaración de una variable define inmediatamente su tipo.

En PASCAL existen cuatro tipos normalizados de datos; enteros, reales, caracteres y booleanos. La notación «escalar» se refiere a que son datos de una única dimensión, o lo que es lo mismo: que sólo definen una característica del elemento al que se refieren. No hay que olvidar que el PASCAL permite además la estructuración de los datos en formas más complejas. Cada tipo de datos tiene su manera particular de expresar las constantes, así como sus propias operaciones y operadores definidos que permiten obtener valores de un tipo determinado con operandos de ese mismo tipo.




*La cabecera de un programa actúa a modo de «etiqueta» que informa al usuario del contenido u objetivo del mismo.*




*En la zona de declaraciones y definiciones de un programa PASCAL hay que diferenciar claramente los datos constantes de los variables.*

## Datos de tipo entero

Son los que presentan a un número entero, esto es: a una secuencia de dígitos decimales precedidos o no por un signo aritmético de suma o resta. En una constante entera no se admiten espacios en blanco, comas, ni puntos.

El delimitador o palabra clave que permite declarar datos de tipo entero es INTEGER. Ejemplo:

```
VAR DIAS, MESES: INTEGER;
```

Cuando se desee evaluar una expresión en la que los operandos son enteros, los siguientes operadores producirán un resultado de tipo entero:

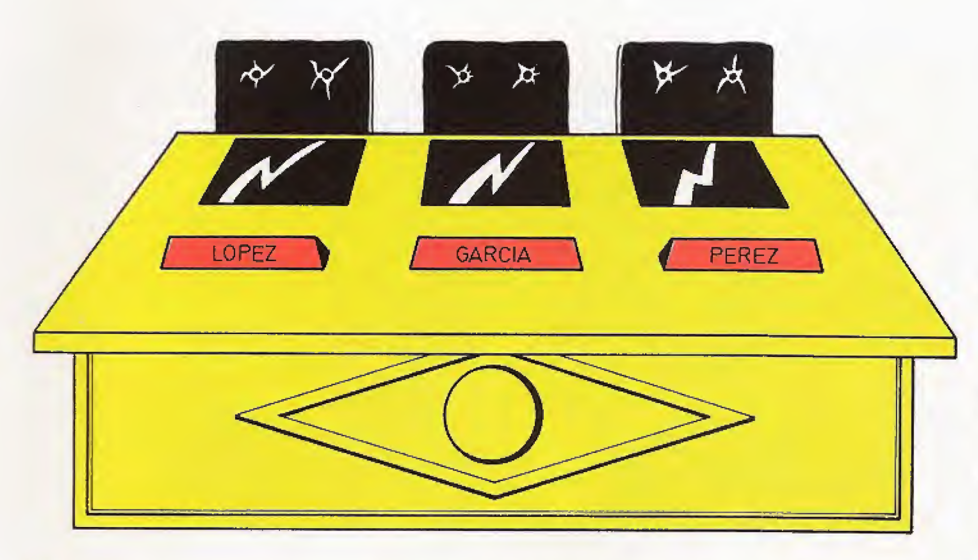
— Suma (+): a+b; suma a y b.

— Resta (–): a–b; resta de a el valor de b.

— Multiplicación (*): a*b; multiplica a*b.

— División (DIV): a DIV b; divide a entre b y trunca los decimales.

— Resto entero (MOD): a MOD b; calcula el resto entero de la división de a entre b.




*Los identificadores sirven para asignar nombres a las variables y, de esta forma, permitir su uso posterior dentro del programa.*

Ejemplos:

16 DIV 4 (el resultado es 4)
7 DIV 3 (el resultado es 2)
7 MOD 3 (el resultado es 1)

## Datos de tipo real

El tipo real define a un elemento del conjunto de los números reales. Estos datos se representan con una parte entera y una parte fraccionaria, separadas por un punto decimal y precedidos o no por el signo correspondiente.

Cuando se quieran representar números reales muy grandes o muy pequeños, es conveniente utilizar la notación científica o exponencial. Esta incluye una «E» seguida por el exponente de la potencia de 10 que corresponda. Por ejemplo, para representar el número un millón se utilizará: 1.0 E6, ya que no se puede omitir la parte entera ni la parte fraccionaria. Por último, cabe señalar que las variables de tipo real se declaran con la palabra REAL.

Por ejemplo:

```
VAR TEMPERATURA: REAL;
```

Si se tiene una expresión en la que al menos uno de los operandos es de tipo real (el otro puede ser de tipo entero), los siguientes operadores obtendrán un valor de tipo real como resultado:

— Suma (+): a+b; suma a con b.
— Resta (-): a-b; resta de a el valor de b.
— Multiplicación (*): a*b; multiplica a por b.
— División (/): a/b; divide a entre b.

En el caso de la división, el resultado será de tipo real aunque los operandos sean de tipo entero.

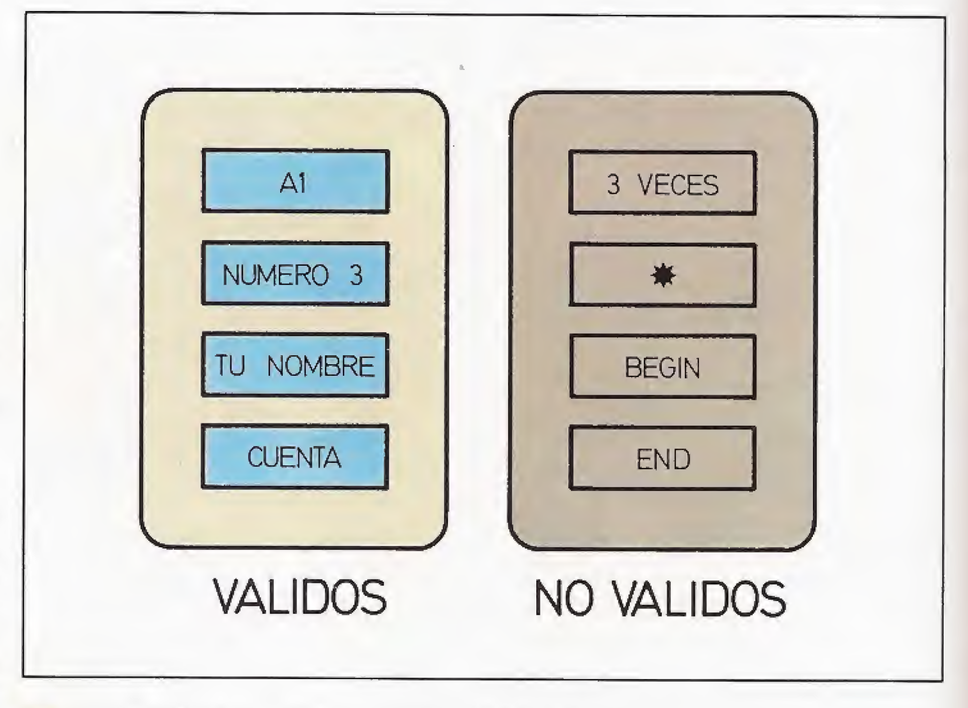
## Datos de tipo carácter

Este tipo de datos consiste en un carácter encerrado entre apóstrofes (' '). Por ejemplo: 'A', '2', '*'... son datos de tipo carácter. Para representar un apóstrofe, éste se escribe dos veces: (' ' ' ').

Cada variable de tipo carácter tomará el valor de un solo carácter; su declaración en el programa se hace por medio de la palabra CHAR.

Ejemplo:

```
VAR LETRA: CHAR;
```




*Los identificadores deben ajustarse a ciertas reglas: por ejemplo, no son utilizables como identificadores las palabras reservadas del PASCAL, o las palabras que empiecen por una cifra numérica.*

Para este tipo de datos no existen operadores específicos que permitan efectuar cálculos con ellos; no obstante, sí se pueden introducir las constantes y las variables de tipo carácter de lectura, escritura y asignación.

En contrapartida, el PASCAL tiene definidas dos funciones estandarizadas que sólo son aplicables a los datos de tipo carácter. Estas son:

— ORD( <car.> ) que proporciona el número ordinal correspondiente al carácter <car.>, y
— CHR( <num.> ) que da como resultado el carácter cuyo número ordinal es <num.>.

Ambas funciones suelen también denominarse de transferencia, por permitir la identificación entre cada carácter y su número ordinal correspondiente.

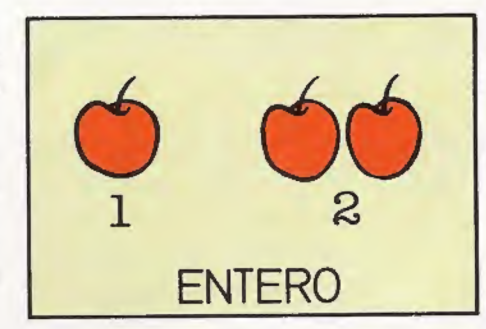
## Datos de tipo BOOLEANO

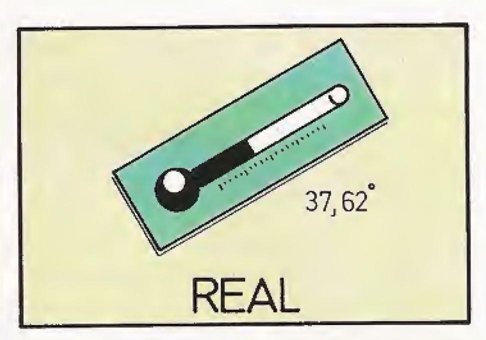
Un dato se dice que es BOOLEANO, o que pertenece al álgebra de Boole, cuando sólo puede tomar dos valores: abierto o cerrado, alto o bajo, cierto o falso, uno o cero...
En PASCAL existen dos palabras reservadas para definir a cada uno de estos valores: TRUE y FALSE (cierto o falso).
Para entenderlo mejor, cabe pensar en qué situación está una ventana: puede estar abierta o cerrada, aunque no es admisible que esté a la vez abierta y cerrada, y tampoco puede estar de otra forma que no sea alguna de ambas. La situación de la ventana será por tanto una variable *booleana*. A una de las situaciones se la asigna el valor TRUE y a la otra FALSE.
Para declarar un dato de tipo BOOLEANO se utiliza la palabra BOOLEAN.
Por ejemplo:

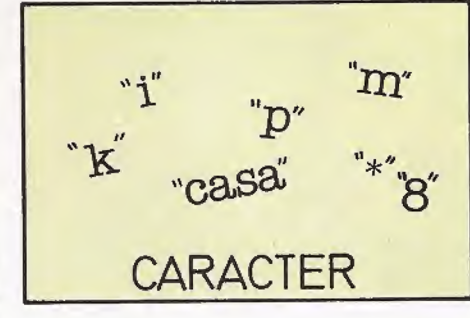
```
VAR CUENTO: BOOLEAN;
```

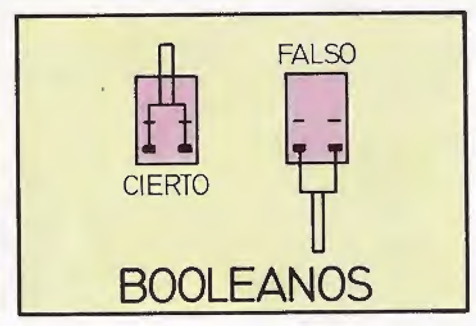
Entre operandos de tipo booleano se pueden aplicar los siguientes operadores que proporcionan un valor de tipo booleano:

— NOT obtiene el valor opuesto al del resto del dato booleano sobre el que se aplica. Realiza la operación lógica «negación».
— AND efectúa el producto lógico; el resultado será FALSE, si al menos uno de los operandos tiene el valor FALSE.
— OR representa la operación de suma lógica; en ella, el resultado será

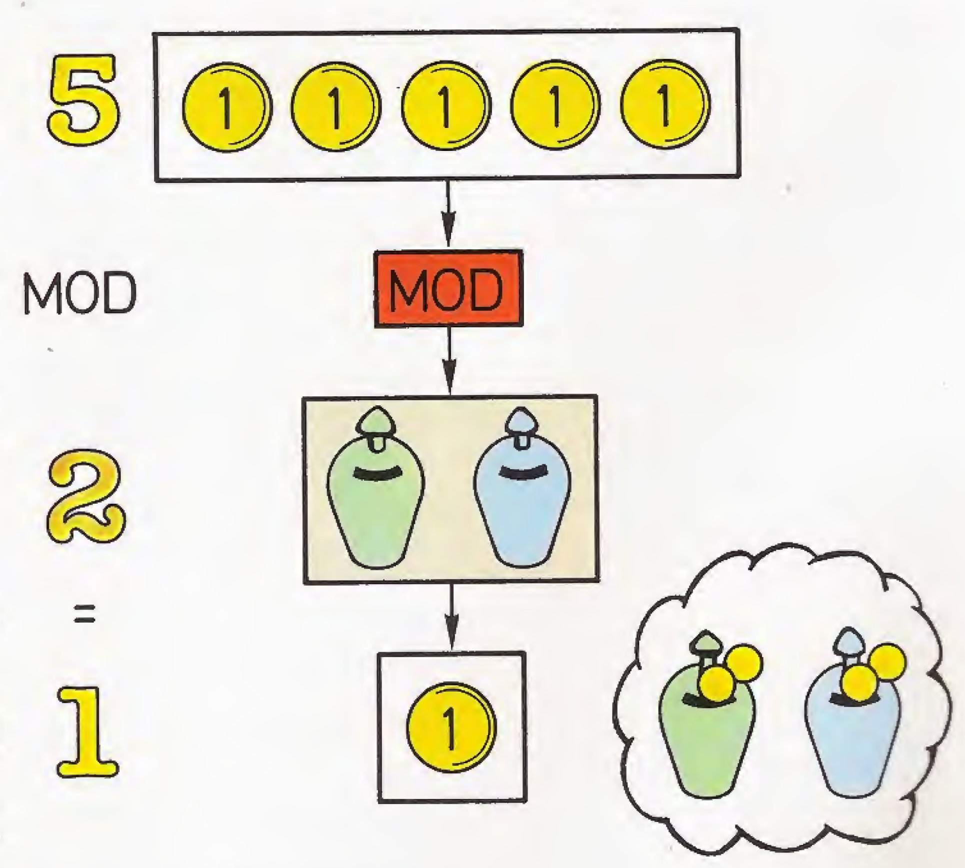


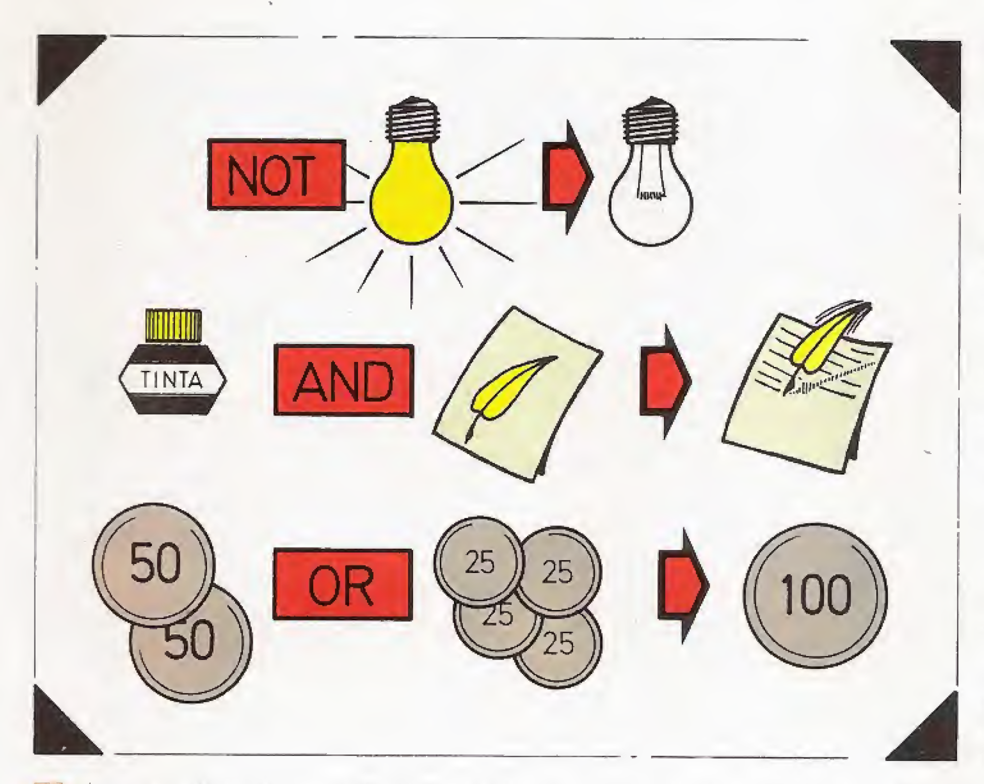




*Las constantes y las variables PASCAL pueden encuadrarse en tres categorías o «tipos de datos»: numéricos (enteros o reales), carácter y booleanos.*




*El operador MOD obtiene el resto de una división entre dos números enteros.*




*Los operadores lógicos NOT, AND y OR manejan variables de tipo booleano; éstas sólo pueden adoptar dos valores: TRUE (cierto) o FALSE (falso).*

TRUE si al menos uno de los operandos tiene el valor TRUE.

Existen además una serie de operadores que se denominan *relacionales,* que obtienen un valor booleano aunque los operandos no sean de tipo booleano:

=: indica la igualdad entre dos expresiones
<>: operador de desigualdad
<: relación «menor que»
>: relación «mayor que»
<=: relación «menor o igual que»
>=: relación «mayor o igual que»

Ejemplos de expresiones booleanas pueden ser:

```
A<>'B'
N>=0
A AND (NOT B)
```

## La sentencia de asignación

Una vez vistos los fundamentos básicos del PASCAL, y estudiado el concepto de variable y su declaración de tipo, hay que analizar la forma en la que pueden utilizarse las variables dentro del contexto del programa.

Cada variable se puede imaginar como una hoja en blanco de un block de notas, en la que se puede ir «anotando» con lápiz un valor constante; éste coincidirá con el valor actual que contiene dicha variable. Posteriormente, es posible «borrar» el contenido de esa hoja del block y volver a anotar otro valor, con lo que se habrá actualizado el contenido de la variable en cuestión.

Estas dos acciones tan sencillas, pero a la vez tan importantes, de asignar un valor a una variable y actualizar ese valor cuando sea necesario, están encomendadas en el PASCAL a la llamada sentencia de asignación, cuya forma general es la siguiente:

```
<variable>: =<expresión>
```

El operador de asignación es el signo ":=", constituido por dos caracteres. Este asigna el valor situado a su derecha, a la variable que le precede. La zona <expresión> puede ser una constante, una variable previamente declarada y con un valor ya asignado, o bien una expresión matemática o lógica; en este último caso, se trataría de una agrupación de constantes y variables unidas por operadores.

La variable será el nombre de un identificador válido PASCAL; por supuesto, declarado como variable en la zona de definiciones y declaraciones del programa.

La sentencia de asignación lleva implícitas dos funciones. La primera es la de obtener un valor al evaluar la expresión situada detrás del operador ":=", y la segunda la de asignar ese valor a la variable que precede al operador de asignación.

Así, por ejemplo:

```
CONTADOR := CONTADOR + 1
ESTAVARIABLE := OTRAVARIABLE
```

En el primer caso, el valor de la variable CONTADOR es incrementado en una unidad y el nuevo valor se asigna a la propia variable CONTADOR, «borrando» así su valor precedente.

Otro cometido importante de la sentencia de asignación es el de comprobar el tipo de variable. Como ya se indicó, cada variable sólo puede ser de un único tipo y, por lo tanto, sólo será aceptada la asignación de una expresión que produzca un valor del mismo tipo que el declarado para la variable.

Hay excepciones a esta regla, puesto que algunos tipos de datos «engloban» a otros. De ahí que, por ejemplo, sea posible asignar un valor entero a una variable declarada como real. En tal caso, será válida dicha sentencia de asignación.

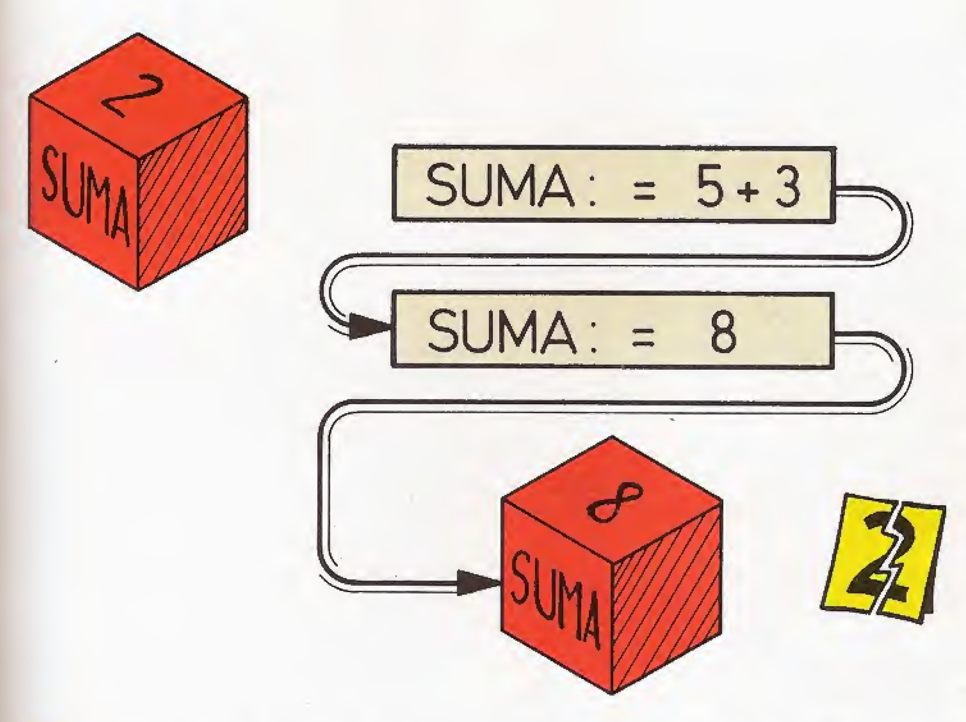
El dialecto de PASCAL más estandarizado no admite la asignación múltiple, o lo que es lo mismo: asignar un mismo valor a una serie de variables. Por ejemplo:

<variable>, <variable2>, ... :=<expresión>

No obstante, algunos compiladores específicos sí permiten tal posibilidad.

## Expresiones aritméticas

El objetivo fundamental de una expresión aritmética es el cálculo de un valor




*Una sentencia de asignación evalúa la expresión incluida en su argumento y otorga el valor resultante a la variable implicada.*

que se utilizará como dato. Este será procesado por el programa cuando sea oportuno. Dicho cálculo se efectuará de acuerdo a una serie de reglas que se exponen a continuación. Las expresiones aritméticas se utilizarán en distintas zonas del programa; por ejemplo, en las sentencias de asignación vistas anteriormente, en la definición de constantes, en el argumento de una función estándar, dentro de la condición impuesta en una sentencia de bifurcación condicional...

Una expresión aritmética consiste en un conjunto de operandos, que pueden ser constantes, variables y/o funciones estándar (seno, logaritmo, raíz cuadrada...), unidos por una serie de operadores aritméticos o lógicos, tales como: «+», «*», «OR», ... Por ejemplo, las siguientes son expresiones aritméticas válidas:

```
SUELDO—((SUELDO*5)/100)
ROUND (PI*R*R)+AREA
```

En el segundo caso, ROUND es una función estándar que obtiene la parte entera redondeada de un número real.

## Jerarquía de los operadores

Una vez conocidos los diferentes operadores utilizables con cada tipo de datos, veamos cómo hay que manejar estos operadores de forma eficiente.

El valor obtenido al evaluar una expresión matemática puede variar según el orden en el que se ejecuten las operaciones indicadas en la misma. Por ejemplo, en la expresión:

```
3 * 6 + 2
```

si primero se ejecuta la operación de multiplicación y después la de suma, el valor que se obtiene es 20. Mientras que si primero se efectúa la operación de suma y después la de multiplicación, el resultado será 24.

Para obviar este problema, los lenguajes de los ordenadores establecen un orden jerárquico de operadores, de tal forma que si no se dispone lo contrario, se efectuarán unas operaciones antes que otras. En el PASCAL, la prioridad y orden jerárquico de los operadores coincide con el siguiente:

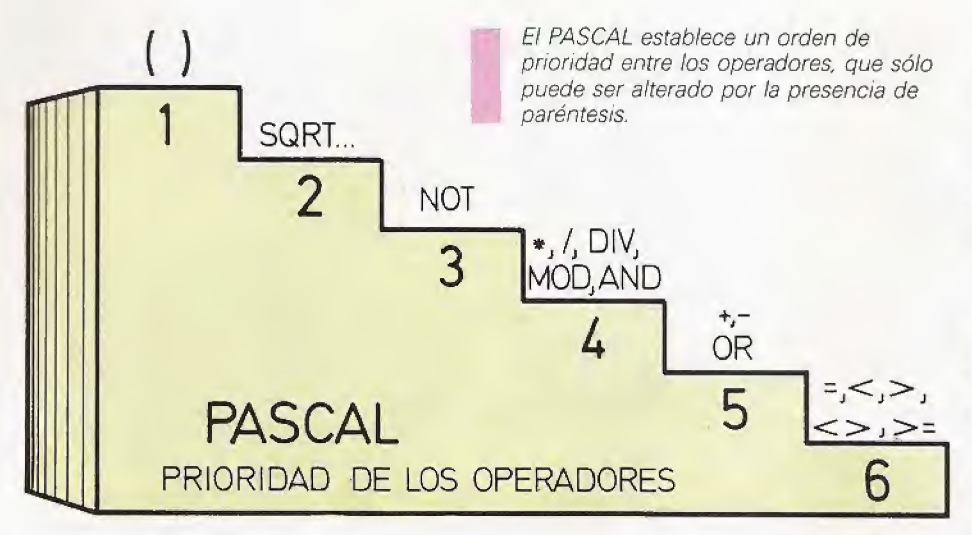
```
1: Funciones estándar
2:NOT
3: *,/,DIV, MOD, AND
4: +,-,OR
5: =,<>,<,>,<=,>=
```




*El contenido de una variable puede ser borrado y actualizado con un nuevo valor por medio de sentencias de asignación.*

Cuando en una expresión aparecen varios operadores con un mismo grado de prioridad, se ejecutará el primero de los que se encuentren situados más a la izquierda.

Esta jerarquía puede ser modificada mediante el uso de paréntesis. Las expresiones encerradas entre paréntesis




*El PASCAL establece un orden de prioridad entre los operadores, que sólo puede ser alterado por la presencia de paréntesis.*

se evaluarán prioritariamente a todas las demás. En el caso de que existan varios paréntesis «anidados», unos dentro de otros, se evaluarán primero los más interiores. Por ejemplo, la expresión siguiente:

((8*2/4)-1)+(5-ORD('A'))

da como resultado el valor 7.

## Ficheros de entrada y salida

La cabecera de un programa PASCAL sirve para dar nombre al programa por medio de un identificador. Este nombre, que sigue a la palabra clave PROGRAM, no tiene un significado especial dentro del contexto del programa; aunque cabe precisar que algunos sistemas particulares pueden asignar a dicho nombre un significado propio fuera del programa para referirse a él específicamente, por ejemplo en orden a su almacenamiento en memorias de masa.

En todo caso, la cabecera del programa tiene otra misión encomendada: declarar una lista de parámetros del programa. La lista de parámetros hace referencia a variables que normalmente serán nombres de ficheros, los cuales deben estar en correspondencia con elementos que ya estén definidos e implantados de alguna forma en el sistema, con independencia del programa.

Cada ordenador tiene definido un conjunto de caracteres (letras, números y signos especiales), que resultan «inteligibles» para el propio ordenador y para los dispositivos de entrada y salida.

Sobre este conjunto de caracteres, el PASCAL tiene definidas dos variables de tipo fichero: INPUT y OUTPUT. Este tipo de variable se declara con la palabra clave FILE. A pesar de que se estudiará con más detalle posteriormente, conviene adelantar que se trata de un conjunto estructurado de datos en el que todos ellos son del mismo tipo.

El PASCAL tiene ya definido un tipo de variable que se declara con la palabra reservada TEXT y que no es más que un fichero de caracteres (FILE OF CHAR). Los ficheros estándar INPUT y OUTPUT son de este tipo, de forma que, para el programa PASCAL, es como si previamente se hubiera realizado la siguiente declaración:

VAR INPUT,OUTPUT: TEXT;

En consecuencia, al escribir un programa no será necesario volver a declarar estas variables, sino únicamente mencionarlas en la lista de parámetros de la cabecera.

INPUT y OUTPUT, como parámetros formales del programa, representan a los dispositivos de entrada/salida de un sistema informático: el teclado, la impresora o la unidad de discos. Gracias a ellos se facilita la comunicación entre un programa PASCAL y el sistema bajo cuyo control se esté ejecutando el programa.

Como los ficheros INPUT y OUTPUT son utilizados muy a menudo por la mayoría de los programas, se consideran como parámetros implícitos en ciertas funciones y procedimientos estandarizados cuando no se especifica lo contrario. Por lo tanto, serán inicializados automáticamente y con antelación a la ejecución del programa, al ser declarados en la cabecera del mismo.

Si en un programa se va a hacer referencia a otros ficheros externos —ya existentes antes de la ejecución del programa, o que van a ser creados durante la ejecución—, los nombres de estos ficheros deben aparecer en la lista de parámetros del programa; además, deben ser declarados en la zona de definiciones y declaraciones como variables de tipo fichero. Por ejemplo:

PROGRAM EJEMPLO (INPUT,OUTPUT, X,ARCHIVO); VAR X,ARCHIVO: FILE OF <tipo>;

siendo <tipo> el tipo de las variables que constituyen ese fichero.

### Breve reseña histórica

Resulta fácil deducir que el nombre de PASCAL, otorgado a este lenguaje, procede del insigne matemático, físico, filósofo y escritor francés Blas Pascal, quien ya en 1641, a sus escasos 18 años inventó una máquina de calcular.

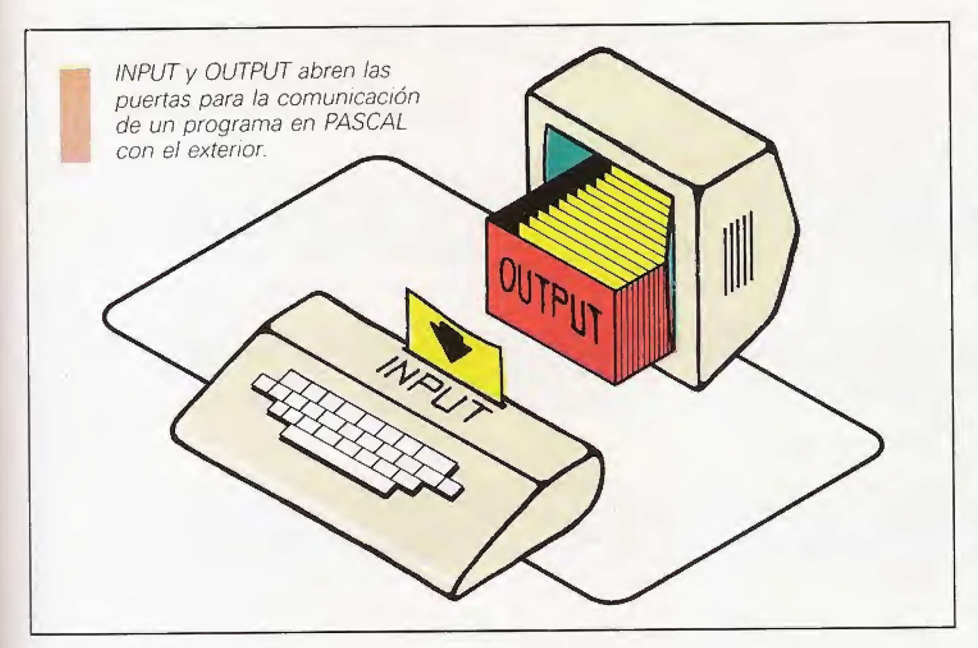
Los orígenes del PASCAL como lenguaje de programación se deben a un grupo de trabajo dirigido por Nicklaus Wirth.

Partiendo de las versiones 68 y W del ALGOL, trataron de obtener un lenguaje de alto nivel, muy sistematizado y pedagógico, adscrito a las tendencias de la programación estructurada.

En 1970 se presentó el primer diseño de compilador. Posteriormente el PASCAL se fue afianzando hasta llegar a estandarizarse en 1978. Actualmente, forma parte, por derecho propio, de los lenguajes habituales en el mundo del software.

## Lectura de datos

Un programa sin intercambio de información con el exterior tendrá poca utilidad al realizar siempre la misma función. En cambio, al poder recibir información del exterior ve aumentada su flexibilidad: proporcionará resultados de salida diferentes cada vez, en función de los datos que reciba. Suponiendo, por ejemplo, que ya se tienen declaradas con anterioridad las variables tensión, resistencia y corriente, como reales, las siguientes sentencias de asignación cal-




*INPUT y OUTPUT abren las puertas para la comunicación de un programa en PASCAL con el exterior.*

cularán un valor para la variable tensión aplicando la ley de Ohm:

```
resistencia: = 100;
corriente: = 3;
tensión: = resistencia*corriente;
```

No obstante, para calcular ahora la tensión si la corriente vale por ejemplo 2, será preciso modificar el programa, incluyendo una nueva sentencia de asignación: corriente: = 2; en lugar de la ya existente, lo que resta efectividad al programa. Para resolver esta situación, hay que utilizar una o más sentencias de entrada de datos, que en PASCAL se codifican mediante el comando READ. Esta sentencia lee un dato del fichero de entrada, especificado en el argumento de la propia sentencia de lectura, y lo asigna a la variable del programa que se incluye a continuación, en la misma sentencia READ.

Si no se especifica la fuente de los datos, se asume como tal al fichero estándar INPUT. En consecuencia, cuando se utiliza este fichero, no es necesario indicarlo expresamente en la sentencia READ. El formato genérico de la sentencia es el siguiente:

```
READ(<fichero>, <var.> <var.>, ...);
```

El campo <fichero> es opcional y coincide con el nombre de la fuente de datos de entrada de la que se van a leer los valores. A su vez, <var> son las variables que recibirán la asignación de los datos leídos.

Así pues, se puede leer un solo dato del fichero estándar INPUT, aunque también se pueden leer varios datos de ese mismo fichero y asignarlos al grupo de variables, separadas por comas que figuren en la sentencia READ; o bien, es posible, asimismo, extraer uno o varios datos para asignarlos a las variables desde cualquier otro fichero externo, que previamente se haya especificado en la lista de parámetros de la cabecera como fuente de los datos de entrada.

Para aumentar la utilidad del ejemplo anterior, pueden sustituirse las dos primeras sentencias de asignación por la siguiente instrucción READ:

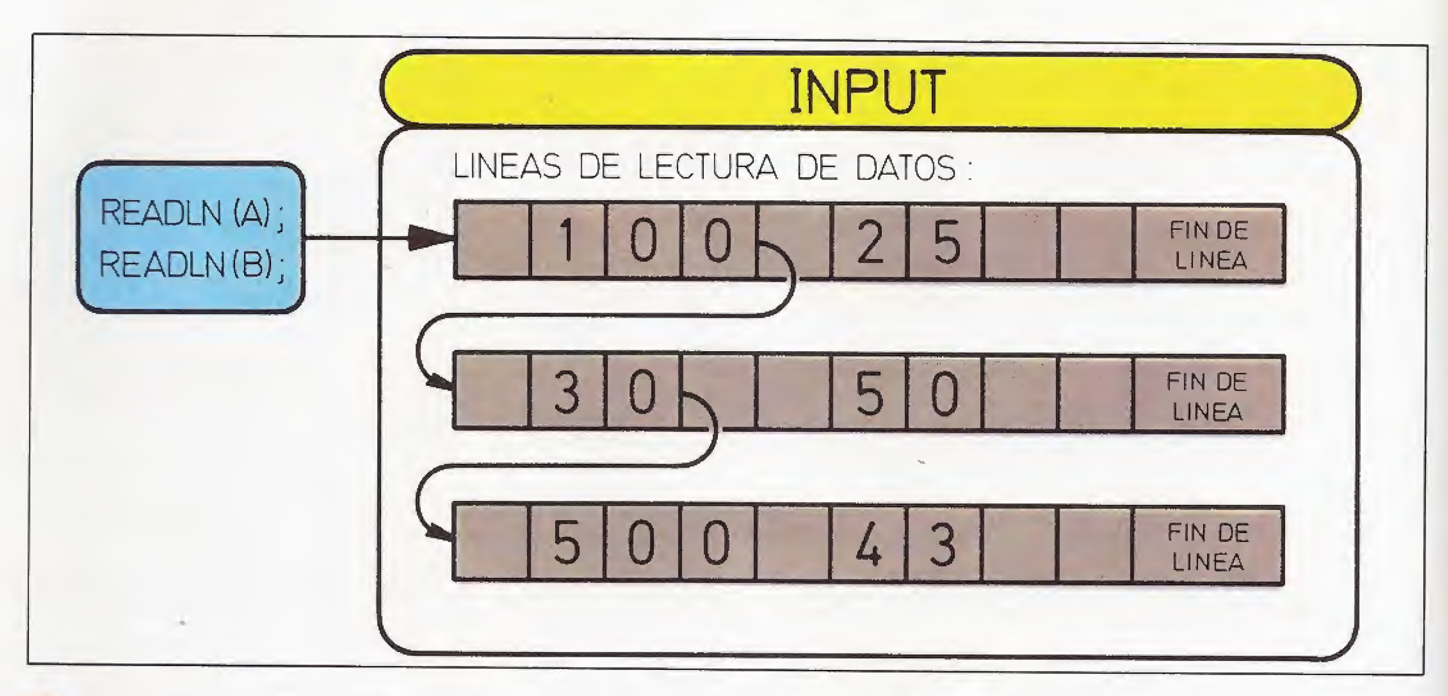
```
READ(resistencia,corriente);
```

Con lo que el programa leerá los datos de resistencia y corriente y obtendrá un resultado dependiente de los datos leídos. En el soporte de entrada, los datos deben aparecer separados por espacios en blanco o por delimitadores de fin de línea; ello difiere, normalmente, para cada compilador PASCAL específico y dependerá del tipo de entrada de datos, esto es: si tiene lugar a través del teclado o por medio de cualquier otro dispositivo como, por ejemplo, un fichero de entrada residente en cinta o en disco magnético.

*La capacidad para interactuar con el mundo exterior es uno de los factores determinantes de la versatilidad de un ordenador, ya que permite la elaboración de distintos datos y la consecución de diversos resultados a partir de un mismo programa.*

Al utilizar la sentencia READ, los datos a leer estarán dispuestos en líneas; de tal forma que cuando se hayan leído todos los datos de una línea, se saltará automáticamente a la línea siguiente. La lectura de los mismos se realizará siempre de izquierda a derecha.

Si se quiere forzar el salto a una nue-




*La sentencia READLN permite una lectura selectiva de los datos de entrada, saltando automáticamente a la línea siguiente tras leer el dato ordenado.*

va línea de entrada, se puede utilizar la sentencia:

```
READLN(<fichero>,<var.>,...);
```

Tal como se observa, su formato es análogo al de la sentencia READ convencional, con la única diferencia de que ahora, después de leer las variables especificadas, se salta a la siguiente línea de entrada, aunque aún no se hayan terminado de leer todos los datos de esa línea. Los datos que sigan al último leído serán ignorados, y la próxima sentencia de lectura leerá el primer dato de la línea siguiente. Así, por ejemplo, si los datos de entrada están distribuidos en dos líneas de la siguiente forma:

```
100 25
2   50
```

la ejecución de las sentencias que siguen asignará el valor 100 a resistencia y saltará a la siguiente línea, ignorando el dato 25, para asignar el valor 2 a la variable corriente:

```
READLN(resistencia);
READLN(corriente);
```




*Para conocer el resultado del proceso realizado por el ordenador es imprescindible utilizar sentencias de salida de datos. Estas proyectarán los datos hacia los periféricos de salida.*

## Salida de datos

El resultado de un programa no suele resultar útil si no es comunicado al exterior a través de algún dispositivo de salida asociado al ordenador. La misión de una instrucción de salida es la de presentar cualquier expresión o valor de una variable en la pantalla o en cualquier otro dispositivo de salida como, por ejemplo, la impresora. En PASCAL, la sentencia especializada en este cometido es WRITE, cuyo formato coincide con:

```
WRITE([<fichero>,]<exp.>[,exp.>,...]);
```

El campo <fichero> define al fichero

de salida de los datos, y su presencia es opcional; si bien, en el caso de incluirse, resultará imprescindible haberlo mencionado en la lista de parámetros de la cabecera, tal y como ocurría con la sentencia READ. La zona <exp.> contiene a las expresiones matemáticas o, simplemente, a las variables cuyo valor se desea representar. En este caso, cuando no se especifique el fichero de destino de los datos de salida, se asumirá como tal al fichero estándar OUTPUT. Continuando con el ejemplo inicial, para que sea posible conocer cuál ha sido el resultado de la ejecución, habrá que añadir una sentencia de salida. El programa completo adoptará el siguiente aspecto:

```
PROGRAM LEYDEOHM (INPUT, OUTPUT);
VAR tensión, resistencia, corriente: REAL;
BEGIN
  READ(resistencia,corriente);
  tensión:=resistencia*corriente;
  WITE(tensión)
END.
```

Como en el caso de READ, en una sentencia WRITE se pueden incluir varios elementos para ser presentados; éstos deben aparecer separados por co-

```
WRITELN (TENSION) ;
WRITELN (RESISTENCIA) ;
WRITELN ;
WRITELN (CORRIENTE) ;
```

```
TENSION = 200
RESISTENCIA = 110

CORRIENTE = 1,818
■
```

*Cuando la salida de datos se ordena a través de una sentencia WRITELN, se producirá un salto automático a la siguiente línea de presentación tras visualizar el dato o mensaje.*

mas. Si los referidos elementos o parámetros son expresiones matemáticas, se evaluarán previamente, y el valor obenido será presentado a través del dispositivo de salida que corresponda. La sentencia WRITE del ejemplo se podría haber escrito como:

```
WRITE(resistencia*corriente);
```

De esta forma, se evaluaría el producto antes de mostrar su valor resultante. Tal y como aparece, el programa ejemplo resulta bastante frío. Si la entrada se realiza a través del teclado, puede realizarse su presentación externa ordenando al programa que indique con un mensaje qué tipo de entrada es la que espera recibir. Para ello se puede utilizar la sentencia.

WRITE ('introducir resistencia y corriente'), precediendo a la instrucción READ, con lo que el literal del paréntesis se reproducirá tal cual en la pantalla.

Cada expresión de la sentencia WRITE puede, opcionalmente, adoptar la forma:

<exp.>:<C1>:<C2>

La zona <C1> es un número natural que indica el número de caracteres con que debe ser escrita la expresión

**TABLA DE COMANDOS PASCAL**

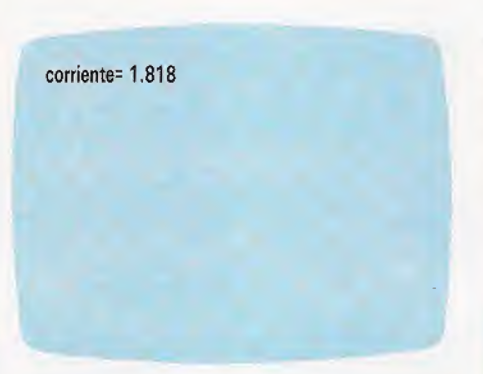
| Instrucción | Cometido | Observaciones |
|---|---|---|
| PROGRAM <id.>(<par.>); | Asignación de parámetros de acceso a un programa | Cabecera del programa |
| BEGIN <ins.>[,<ins.>,...]END; | Delimita bloques de instrucciones ejecutables | Cuerpo del programa |
| CONST <id.>=<valor>;[<id.>=<valor>;][ ... ] | Definición de constantes | Parte de las declaraciones y definiciones |
| VAR <id.>[,<id.>,...]: <tipo>; [ ... ] | Declaración de variables y su tipo | Parte de las declaraciones y definiciones |
| INTEGER | Declaración del tipo entero | Zona de declaraciones y definiciones |
| REAL | Declaración del tipo real | Zona de declaraciones y definiciones. |
| CHAR | Declaración del tipo carácter | Zona de declaraciones y definiciones |
| BOOLEAN | Declaración del tipo booleano | Zona de declaraciones y definiciones |
| INPUT | Declarar fichero estándar de entrada | Cabecera del programa |
| OUTPUT | Declarar fichero estándar de salida | Cabecera del programa |
| TEXT | Declaración de tipo fichero de caracteres | Declaraciones y definiciones |
| READ({<fich.>}<var.>{,...}); | Lectura de datos de un fichero | Bloque de sentencias |
| READLN({<fich.>}<var.>{,...}); | Lectura de datos y cambio de línea | Bloque de sentencias |
| WRITE ({<fich.>}<exp.>{,...}); | Escritura de expresiones | Bloque de sentencias |
| WRITELN ({<fich.>}<exp.>{,...}); | Escritura de expresiones y cambio de línea | Bloque de sentencias |

<id.>: identificador válido PASCAL. <par.>: lista de parámetros de los ficheros que utilizará el programa. <ins.>: instrucción ejecutable. <valor>: valor numérico o literal constante. <tipo>: uno de los tipos de datos normalizados o previamente definidos en PASCAL. <fich.>: fichero declarado con anterioridad. <var>: variable ya declarada. <exp.>: expresión válida.

<exp.>. En el caso de que <exp.> necesite menos caracteres, se completará la expresión con espacios en blanco por la derecha. <C2> es otro número natural que revela el número de dígitos de que constará la parte fraccionaria de <exp.> en el caso de que ésta sea de tipo REAL. Así, por ejemplo, las siguientes líneas:

```
READ(tensión, resistencia);
corriente:=tensión/resistencia;
WRITE('corriente=',corriente:7:3);
```

producirán la siguiente salida:




desde luego, suponiendo que el valor de la tensión es 200 y el de la resistencia 110.

Como se observa, la sentencia WRITE presenta los datos en una única línea. Si se desea presentarlos en varias líneas será preciso utilizar una sentencia WRITELN. Esta comparte el mismo formato que WRITE, si bien, después de escribir las expresiones que se le indiquen, provocará un salto a la siguiente línea de salida:

```
WRITELN('tensión=',tensión:4);
WRITELN('resistencia=',resistencia:4);
WRITELN;
WRITE('corriente=',corriente:7:3);
```

El bloque de sentencias anterior producirá el siguiente resultado en la pantalla:

```
tensión= 200
resistencia= 110

corriente= 1,818
```

En efecto, se observa que la sentencia WRITELN sin parámetros genera como resultado la escritura de una línea en blanco.

## Aspecto general de un programa

```
PROGRAM RAICES2 (INPUT,OUTPUT); (*CABECERA*)
CONST DOS=2; (*DECLARACIONES Y DEFINICIONES*)
VAR A,B,C.RAIZ1,RAIZ2,IMAG,PREAL, DISCRI:REAL;
BEGIN (*CUERPO PRINCIPAL*)
    READ (A,B,C)
    IF A=0 THEN WRITE ('ERROR')
    ELSE
    BEGIN
        DISCRI:=SQR(B)-4*A*C;
        IF DISCRI<0 THE
            BEGIN
                DISCRI:=-DISCRI;
                IMAG:=SQRT(DISCRI)/(DOS*A);
                PREAL:=-B/(DOS*A);
                WRITELN('RAIZ1=',PREAL,'+J',IMAG);
                WRITELN('RAIZ2=',PREAL,'-J',IMAG)
            END
        ELSE
            BEGIN
                RAIZ1:=(-B+SQRT(DISCRI))/(DOS*A);
                RAIZ2:=(-B-SQRT(DISCRI))/(DOS*A);
                WRITELN('RAIZ1=',RAIZ1);
                WRITELN('RAIZ2=',RAIZ2)
            END
    END
END.
```

Un programa escrito en lenguaje PASCAL se puede dividir en tres zonas fundamentales:

- *La cabecera* que da nombre al programa y le asigna una lista de parámetros. Estos coinciden con los nombres genéricos de los ficheros que se utilizan en el programa para el intercambio de información con el usuario (entrada y salida de datos).

- *Declaraciones y definiciones.* Zona en la que se reseñarán todas las etiquetas, constantes, tipos, variables, procedimientos y funciones a utilizar en el resto del programa.

- *El cuerpo del programa* que, como su nombre indica, constituye el conjunto de instrucciones ejecutables que realizarán la función asignada al programa.

En el ejemplo se observa la puesta en práctica del referido *formato libre;* en efecto, la indentación contribuye a mejorar la presentación del programa.
El formato libre implica además la presencia de un separador de instrucciones, coincidente con el signo punto y coma (;). Puede conducir a confusión el hecho de que el ejemplo incluya sentencias que no acaban en punto y coma.
Ello es correcto y se debe a que alguna de ellas conforman una única instrucción, como se verá más adelante; en consecuencia, sería un error fraccionarlas con algún punto y coma.

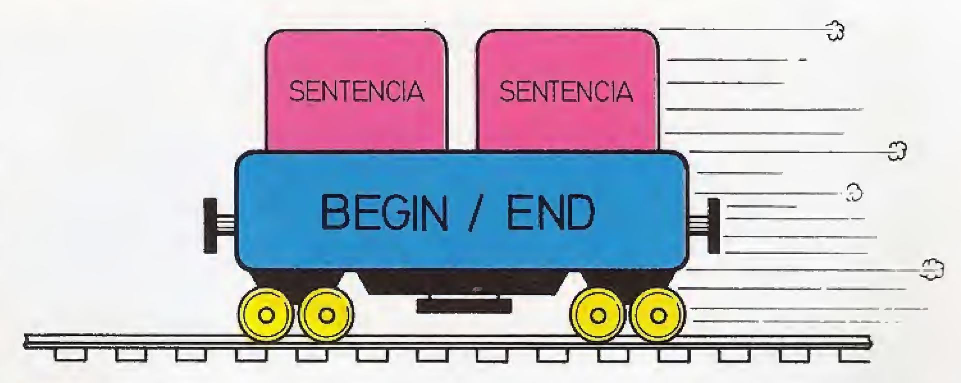
# PASCAL (2)

## Estructuras y sentencias de control. Procedimientos



La programación estructurada aporta un conjunto de técnicas que permiten una fácil depuración y puesta a punto de los programas. Estos resultan inteligibles para otras personas distintas del programador y susceptibles de recibir posteriores modificaciones.

Al igual que ocurre con todos los lenguajes de programación, un programa en PASCAL se puede codificar por medio de una sucesión de sentencias que se van ejecutando ordenadamente, una tras otra, siguiendo las directrices de lo que se llama *flujo del programa.* La continuidad del flujo del programa está garantizada en el PASCAL por su carácter de lenguaje estructurado. Ello hace que sea posible preservar la secuencia de ejecución y, por lo tanto, que no se distraiga la atención intentando comprender la lógica del programa siguiendo los saltos incondicionales de una a otra parte del mismo.




*Una sentencia compuesta, delimitada por las palabras clave BEGIN y END, puede englobar múltiples sentencias elementales. A pesar de ello, la máquina trata a las sentencias compuestas como si se tratara de sentencias únicas.*

### Estructura secuencial

Recordando los conceptos de la programación estructurada, la primera estructura que nos encontramos es la *secuencial.* En el PASCAL, una estructura secuencial está formada por sentencias de acción, que no desvían el flujo del programa; tal es el caso de las sentencias de asignación, de entrada y salida de datos...

El programa en su conjunto se puede considerar como una sentencia única, al estar englobado el cuerpo del programa entre las palabras reservadas BEGIN y END.

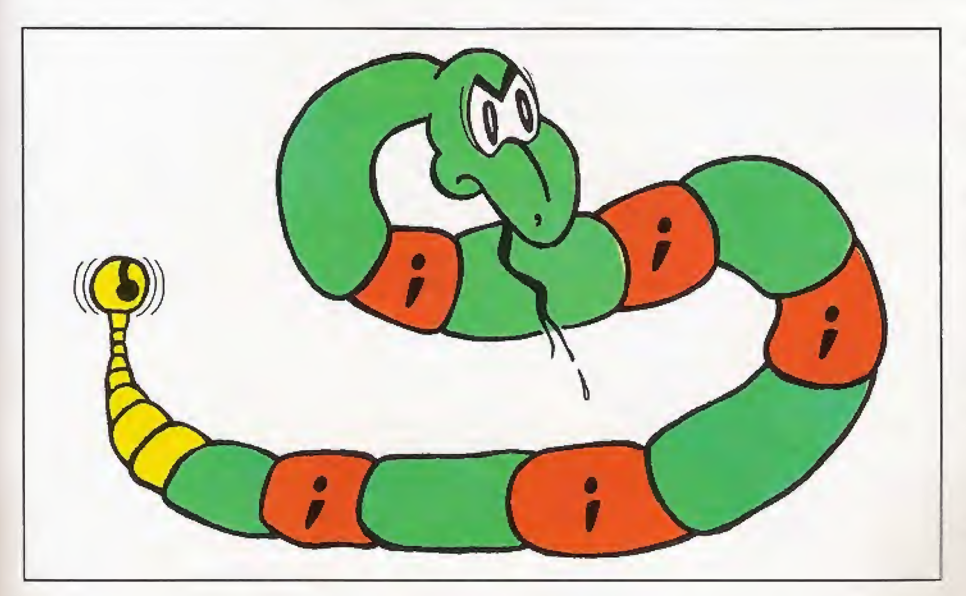
El bloque BEGIN/END conforma una sentencia compuesta, de manera que las sentencias encerradas en su interior se consideran como una sola. Esta es una estructura muy utilizada, como se verá más adelante.

Algunas estructuras de control necesitan llevar asociada una única sentencia y, sin embargo, la propia lógica del programa exige la presencia de varias de ellas para que sea posible programar la tarea implicada; en tal caso, al incluir a todas las sentencias necesarias en un único bloque BEGIN/END, la estructura de control «creerá» que sólo tiene una sentencia asociada.

El uso de sentencias compuestas dentro de un programa debe limitarse exclusivamente a los casos en los que sea plenamente necesario. Aunque el abuso de ellas no produzca ningún tipo de error, es evidente que su presencia innecesaria puede conducir a confusiones, como en el siguiente ejemplo:

```
BEGIN
   BEGIN
         READLN(dato1,dato2)
   END;
   BEGIN
         WRITELN('suma=',dato1+dato2);
         WRITELN('resta=',dato1-dato2);
   END
END.
```

Es necesario resaltar de nuevo la obligatoriedad del punto y coma «;» al final de cada sentencia, dada su función de separador. No obstante, en casos como el del ejemplo anterior, puede ser omitido si a continuación va la palabra END. Hay que recordar que END tiene tam-



*En PASCAL, el punto y coma se utiliza como separador de sentencias, mientras que el punto señala el final del programa.*






*El flujo de un programa en lenguaje PASCAL es gobernado por las denominadas estructuras de control. Las dos estructuras básicas son la «secuencial» y la «repetitiva» o cíclica.*

bién un carácter de separador de sentencias. En todo caso, la presencia redundante del punto y coma delante del END no conduce a error alguno.

El punto final «.», localizado detrás del último END, es el que señala el final del programa; si en su lugar apareciera un «;», el ordenador aguardaría a una nueva instrucción, dado que el punto y coma denota únicamente el punto de separación entre dos sentencias.

## Estructuras repetitivas

En condiciones normales, las sentencias se van ejecutando ordenadamente, y una sola vez cada una de ellas, hasta llegar al final del programa. Sin embargo, en muchas ocasiones es necesario que algunas sentencias se ejecuten repetidamente un determinado número de veces, constituyendo lo que se denomina un «bucle». Este tipo de sentencias, denominadas iterativas o repetitivas, se ejecutan bajo el control de una condición impuesta, cuyo cumplimiento o no determina su repetición.

El PASCAL cuenta con tres sentencias iterativas: WHILE/DO, REPEAT/UNTIL y FOR/TO/DO. La primera de ellas tiene el siguiente formato:

```
WHILE <expresión> DO <sentencia>;
```

La <expresión>, de tipo booleano, es la que determina el número de veces que se repetirá la <sentencia>; concretamente, ésta se ejecutará siempre que la <expresión> sea cierta (valor lógico TRUE). Hay que prever que la variable de control de la <expresión> modifique su valor de forma apropiada dentro del bucle; en caso contrario, se podría caer en un bucle infinito, sin final, error muy difícil de detectar éste.

Para comprender mejor el funcionamiento de una estructura repetitiva, veamos el siguiente ejemplo:

```
BEGIN
    READ(n)
    c=1;prod=1;
```







*La sentencia WHILE/DO da pie a una estructura reiterativa. En ella se repite una acción determinada mientras se cumpla la condición impuesta.*

```
    WHILE c<=n DO
        BEGIN
                prod=prod*c;
                c=c+1
        END;
    WRITE('producto=',prod)
END.
```

Suponiendo previamente declaradas las variables que intervienen, este segmento de programa escribirá el producto de los «n» primeros números naturales o, lo que es lo mismo, el llamado factorial de «n». Después de obtener el dato «n» y de inicializar los valores de «c» y «prod», se comprueba si es cierta (TRUE) la expresión: c<=n. Si es así, se ejecutará la sentencia que sigue al DO. En este caso se trata de una sentencia compuesta que engloba a otras dos elementales; técnica ésta muy utilizada como ya se comentó para «engañar» al DO y poder repetir la ejecución de varias sentencias agrupadas. Una vez ejecutada esta sentencia se vuelve a comprobar la condición; si es cierta, se volverá a ejecutar dicha sentencia. Y así sucesivamente hasta que la expresión produzca el valor FALSE, con lo que no se ejecutará la sentencia asociada al DO, sino la siguiente: en este caso, el WRITE que escribe el resultado.

Hay que hacer notar que el bucle no se ejecutará ninguna vez si la expresión produce el valor lógico FALSE al evaluarla por vez primera.

No ocurre así con la estructura REPEAT/UNTIL. En ella, sea cual fuere el valor de la expresión en la primera pasada, se ejecutará primero la sentencia, para pasar después a evaluar la condición. Esto significa que siempre se ejecutará al menos una vez la sentencia interior al bucle.

El formato de la sentencia que da cuerpo a una estructura de este tipo es la siguiente:

```
REPEAT<sentencia> UNTIL <expresión>;
```

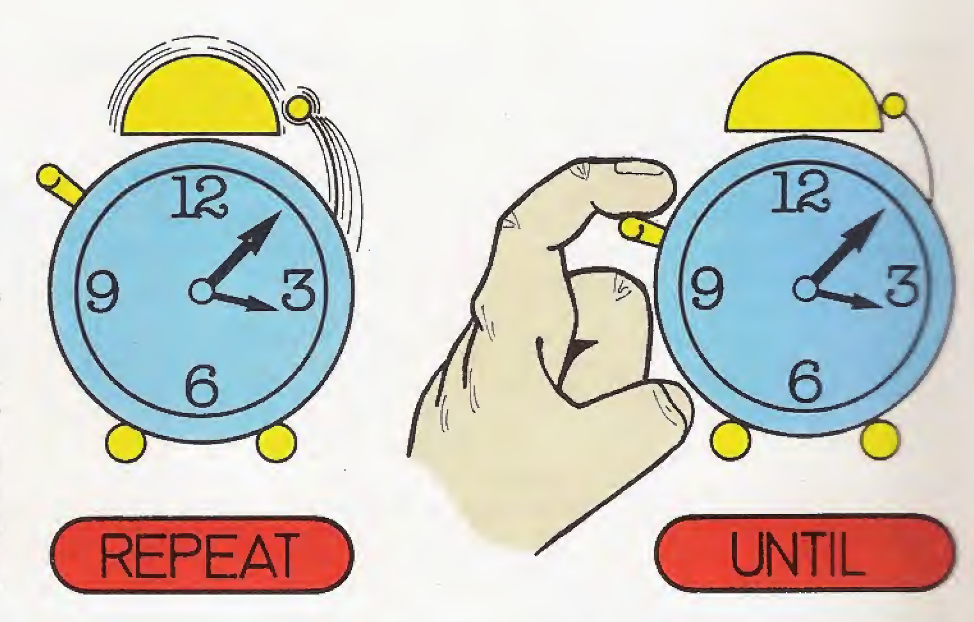
La zona <sentencia>se repetirá (REPEAT) un número indeterminado de veces, hasta que (UNTIL) la <expresión> sea cierta (TRUE). En ese momento se seguirá ejecutando la próxima sentencia del programa, ajena al bucle.

El mismo programa capaz de calcular el producto de los «n» primeros números naturales, se podría codificar con la sentencia REPEAT/UNTIL de la siguiente forma:

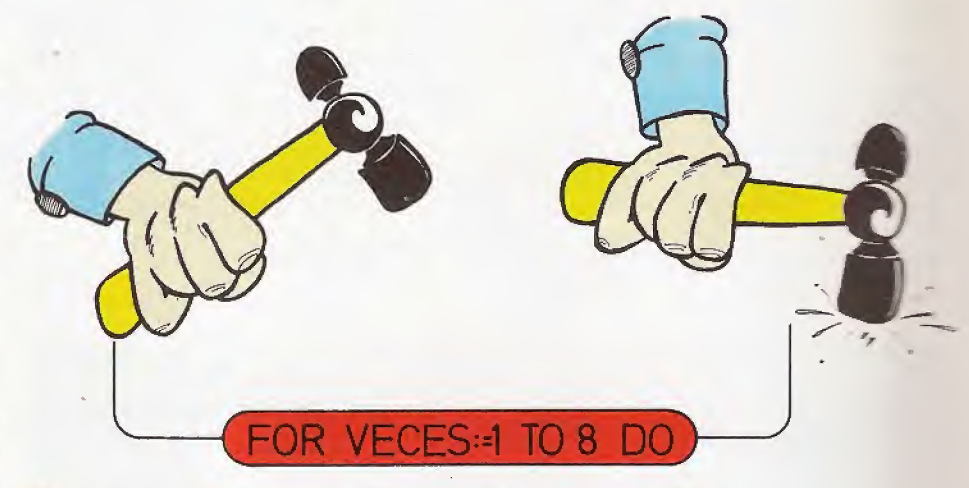
```
    BEGIN
        READ(n);
            prod=1; c=1;
        REPEAT
            prod=prod*c,
            c=c+1
        UNTIL c>n;
WRITE('producto=',prod)
    END.
```

En este caso varía la condición respecto a la del ejemplo que utiliza la sentencia WHILE/DO, debido, precisamente, a la diferencia entre ambas estructuras en cuanto a ejecutar una vez al menos las sentencias del bucle. Otra de las divergencias entre ambas estructuras reside en que ahora no es necesario introducir en un bloque BEGIN/END a todas las sentencias que constituyen el bucle, ya que éstas quedan delimitadas por las palabras REPEAT y UNTIL.

Queda por ver aún una tercera estruc-




*La sentencia o grupo de sentencias encerrada entre las palabras REPEAT y UNTIL se ejecutará repetidamente hasta que se verifique cierta condición.*



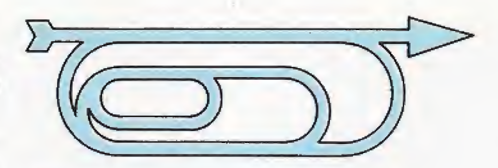

*La sentencia FOR/TO/DO repite una sentencia o grupo de sentencias el número de veces que se indique en su cabecera.*

tura repetitiva, la plasmada por la sentencia FOR/TO/DO. Su principal diferencia con las dos anteriores reside en la funcionalidad de la condición. En las estructuras WHILE/DO y REPEAT/UNTIL, el bucle o conjunto de sentencias se ejecuta reiteradamente hasta que se verifique la condición impuesta, instante en el que se abandonará el bucle. Sin embargo, en el caso de FOR/TO/DO, se repetirá el bucle tantas veces como valores intermedios haya entre los valores inicial y final de la variable de control del bucle. El formato es el siguiente:

```
FOR <variable de control>:=<variable o valor inicial>
TO <variable o valor final >DO<sentencia>;
```

En esta sentencia, tanto la variable de control como sus valores inicial y final, deben ser del mismo tipo y, específicamente, de un tipo ordinal: INTEGER, CHAR, o cualquier otro tipo ordinal declarado por el usuario, como los enumerativos y los subrango, que se verán más adelante. La variable de control no puede ser nunca del tipo REAL.

El valor de esta variable de control no debe ser alterado por las sentencias del bucle. Los valores inicial y final pueden ser incluso expresiones del mismo tipo que la variable de control; expresiones que serán evaluadas antes de entrar en

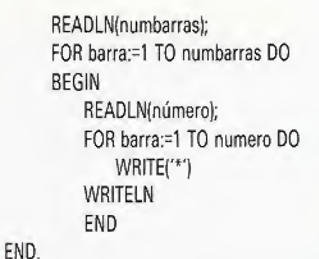


*El anidamiento de sentencias de control permite crear estructuras de mayor complejidad.*

el bucle. El funcionamiento de la estructura FOR/TO/DO es el siguiente. En principio, se asigna a la variable de control el valor inicial y se chequea si ya se ha llegado al valor final, en cuyo caso no se ejecutaría el bucle ninguna vez. Si no alcanza aún el valor final, se ejecutará el bucle y, seguidamente, se incrementará el valor de la variable de control antes de ser comprobado de nuevo. Este proceso se repite hasta que la variable de control adopta el valor final, pasando entonces a ejecutar la próxima sentencia del programa.

El ejemplo que sigue dibujará un histograma, o diagrama de barras, con tantos asteriscos por barra como indique el valor de «número», introducido a través del teclado:

```
PROGRAM asteriscos (INPUT,OUTPUT);
VAR contador,número, barra, numbarras:INTEGER;
BEGIN
    WRITELN('Cuantas barras?');
```



```
    READLN(numbarras);
    FOR barra:=1 TO numbarras DO
    BEGIN
        READLN(número);
        FOR barra:=1 TO numero DO
            WRITE('*')
        WRITELN
        END
END.
```

Como quiera que el bucle más interno está formado por una única sentencia, no es necesario incluirla en un bloque BEGIN/END; no obstante, sí hay que hacerlo con el bucle más externo que consta de más de una sentencia.

La variable de control asociada a esta sentencia puede evolucionar con valores ascendentes o descendentes. Para hacer que «cuente hacia atrás», sólo hay que sustituir la palabra TO por DOWNTO:

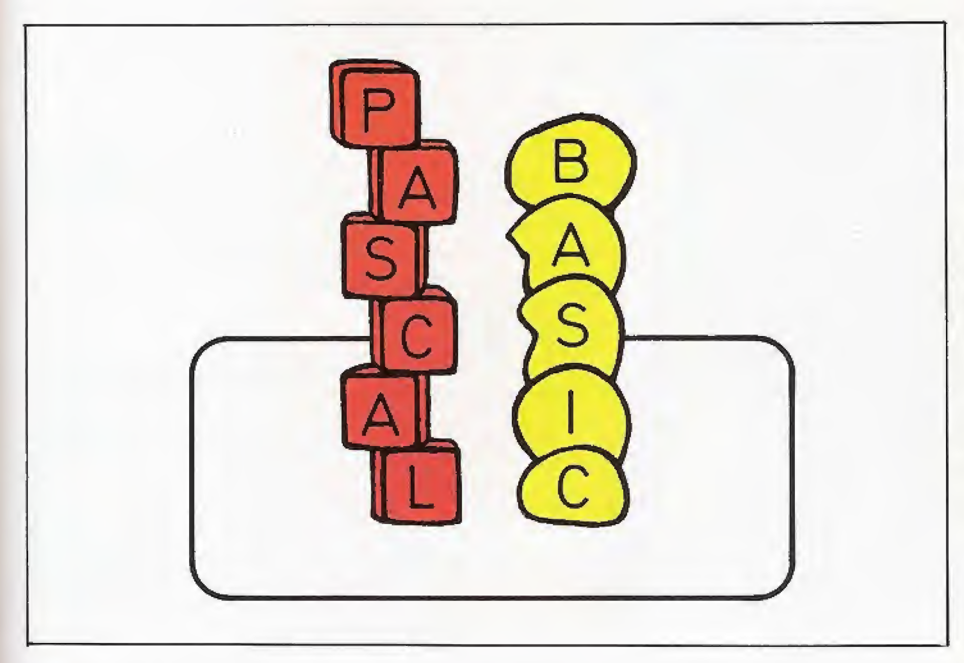
```
PROGRAM asteriscos (INPUT,OUTPUT);
VAR contador,numero, barra,numbarras:INTEGER;
BEGIN
    WRITELN('Cuantas barras?');
    READLN(numbarras);
    FOR barra:=numbarras DOWNTO 1 DO
      BEGIN
          READLN(numero);
          FOR contador:=numero DOWNTO 1 DO
          WRITE('*');
          WRITELN
      END
END.
```

## Sentencias anidadas

Para tener un control completo del flujo del programa, se utilizan las sentencias de repetición y las sentencias de selección que más adelante serán objeto de estudio. Todas ellas incluyen otras sentencias de control gobernadas por la que las contiene. De esta forma se pueden crear estructuras de cualquier nivel de complejidad, «anidando» unas sentencias en otras. El tratamiento que se hace de las sentencias compuestas, por parte del usuario, coincide con el de las sentencias simples. Esta característica otorga al PASCAL una gran legibilidad, facilitando la comprensión de los programas de un simple vistazo.



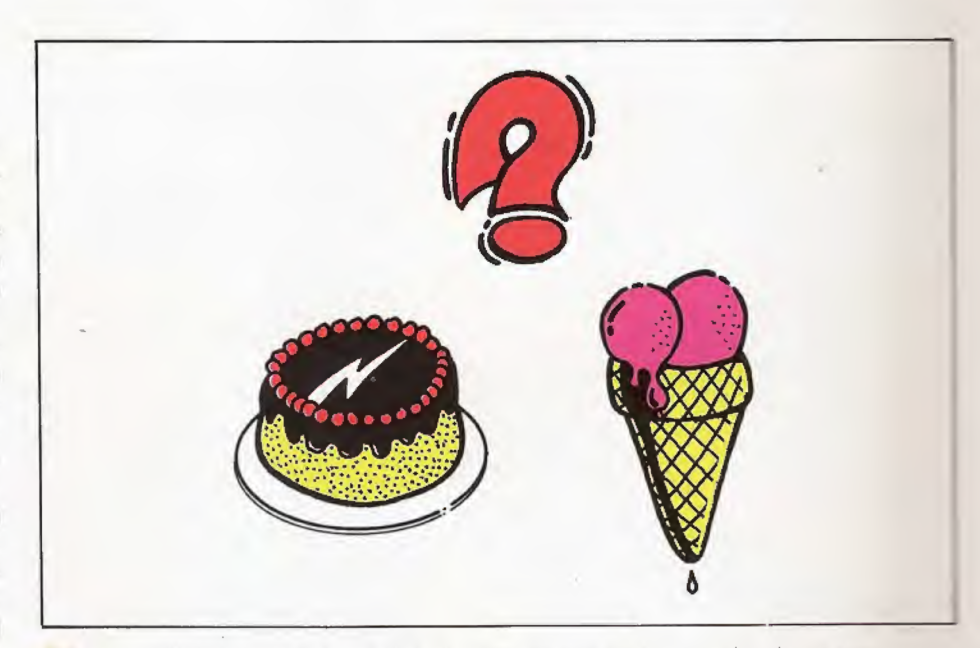

*En la actualidad existen compiladores de PASCAL para todos los ordenadores más populares.*




*La modularidad del PASCAL permite estructurar los programas de forma limpia y clara, facilitando su inteligibilidad y, por ende, su mantenimiento.*

## La sentencia IF/THEN/ELSE

Para concluir con el estudio de las estructuras fundamentales de la programación estructurada, hay que examinar las denominadas estructuras selectivas o condicionales. Su especialidad es la de romper el flujo del programa en un determinado punto del mismo, dependiendo de si se cumple o no una condición establecida. El valor lógico de la condición se comprueba durante el desarrollo del programa. Si es cierto (valor lógico TRUE) se ejecutará una sentencia específica, mientras que si se obtiene el valor lógico falso (FALSE), será otra la sentencia a ejecutar a continuación. La estructura selectiva está fielmente reflejada en la sentencia de control IF/THEN/ELSE. Esta adopta en el PASCAL un formato más elegante y claro que en otros lenguajes, como es el caso del FORTRAN o el BASIC. En efecto, ahora no es necesario utilizar números de línea a los que dirigir la ejecución del programa, sino que es posible introducir hasta programas completos como alternativa a cada una de las decisiones derivadas de la condición impuesta. Su formato general es:



*Las sentencias selectivas ayudan al ordenador a tomar complejas decisiones.*

```
IF <expr.> THEN <sent.> [ ELSE <sent.> ];
```

En ella, <expr.> equivale a una expresión de tipo booleano, cuyo resultado puede adoptar los valores TRUE o FALSE. A su vez, <sent.> corresponde a sentencias simples o compuestas, en cuyo caso deben ir agrupadas dentro de bloques BEGIN/END.

La zona ELSE puede ser omitida; en tal situación, se pasará a ejecutar directamente la sentencia posterior a la asociada a la zona THEN cuando no se verifique la condición impuesta.

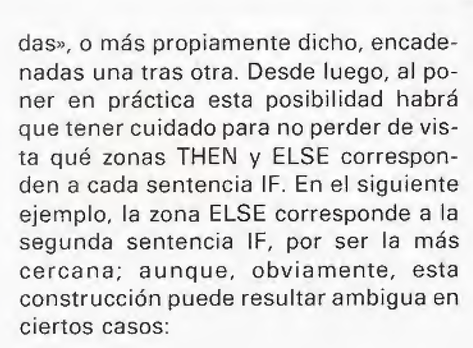
Al tropezar con una sentencia de esta categoría, es fácil imaginar cuál será la actuación del ordenador. Si la condición es cierta, pasará a ejecutar la sentencia correspondiente a la zona THEN, ignorando la asociada a la parte ELSE; por el contrario, si la condición es FALSE, emprenderá la acción inversa. Así, por ejemplo, en el siguiente segmento de programa:

```
IF cierto
    THEN WRITELN('VERDADERO')
    ELSE WRITELN('FALSO');
```

«cierto» será una variable declarada previamente como de tipo booleano cuyo valor actual será visualizado en la pantalla. Una precisión relativa a esta sentencia, es que no hay que colocar un punto y coma «;» delante de la palabra ELSE, aun en el caso de que cada zona de la sentencia IF se escriba en líneas separadas.

Las sentencias IF pueden ser «anida-






*La sentencia IF permite elegir entre dos opciones en función del cumplimiento o no de cierta condición impuesta.*

das», o más propiamente dicho, encadenadas una tras otra. Desde luego, al poner en práctica esta posibilidad habrá que tener cuidado para no perder de vista qué zonas THEN y ELSE corresponden a cada sentencia IF. En el siguiente ejemplo, la zona ELSE corresponde a la segunda sentencia IF, por ser la más cercana; aunque, obviamente, esta construcción puede resultar ambigua en ciertos casos:

```
IF cont>10 THEN IF suma=0
THEN bal :=TRUE ELSE
bal :=FALSE;
```

Para eliminar tal ambigüedad, se puede proceder de dos maneras. Una posibilidad es utilizar la indentación, admitida por el formato libre; ello supone alinear en la misma columna las zonas THEN y ELSE correspondientes a una misma sentencia IF. La segunda alternativa es englobarlas dentro de bloques BEGIN/END de la siguiente forma:

```
IF cont>10 THEN BEGIN IF suma=0
THEN bal :=TRUE ELSE
bal :=FALSE END;
```

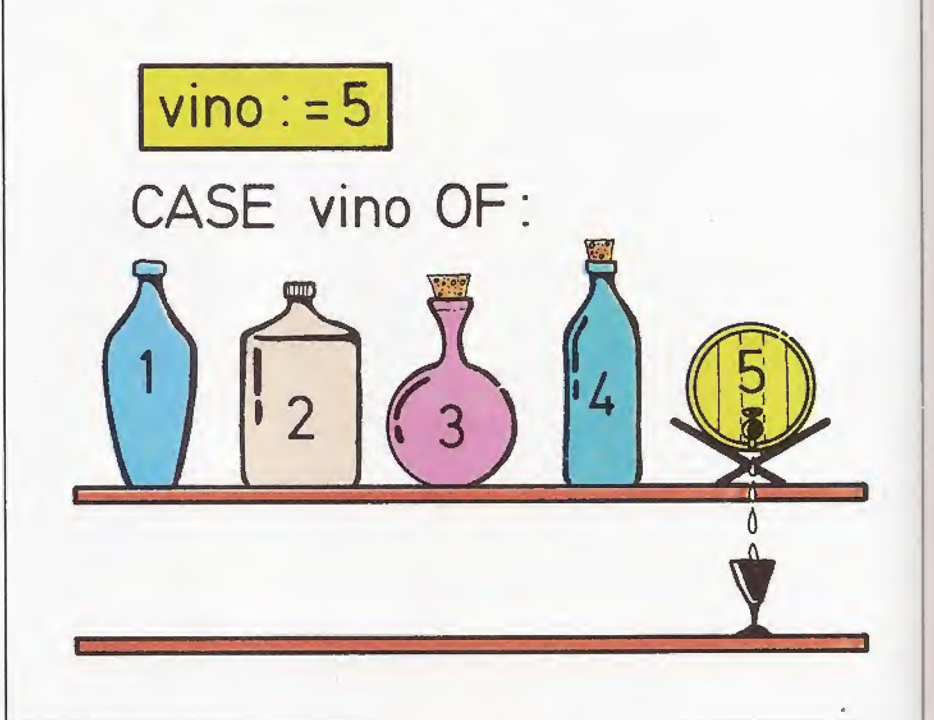
Cabe también la posibilidad de utilizar la siguiente formulación:

```
IF cont>10 THEN BEGIN IF suma=0
THEN bal :=TRUE END ELSE
bal :=FALSE;
```

si lo que se desea es que la zona ELSE, corresponda a la primera sentencia IF. Desde luego, con este método queda más claro a qué sentencias pertenece cada ELSE.

La aplicación más frecuente de la sentencia IF/THEN/ELSE es la de seleccionar una entre varias opciones. Veamos cómo se realiza en el siguiente ejemplo:

```
IF numero>9999 THEN
    cifras:=5
ELSE IF numero >999 THEN
    cifras:=4
ELSE IF numero>99 THEN
    cifras:=3
```




*La estructura CASE/OF brinda la posibilidad de programar tomas de decisión en las que hay que optar por una entre más de dos alternativas.*

```
ELSE IF numero>9 THEN
    cifras:=2
ELSE cifras:=1;
```

Esta construcción permite ejecutar la primera sentencia IF cuya condición sea cierta, abandonándose tras ello toda la estructura selectiva, de tal forma que sólo se ejecutará una de las sentencias de asignación incluidas en la misma.

## Muestrario para elegir

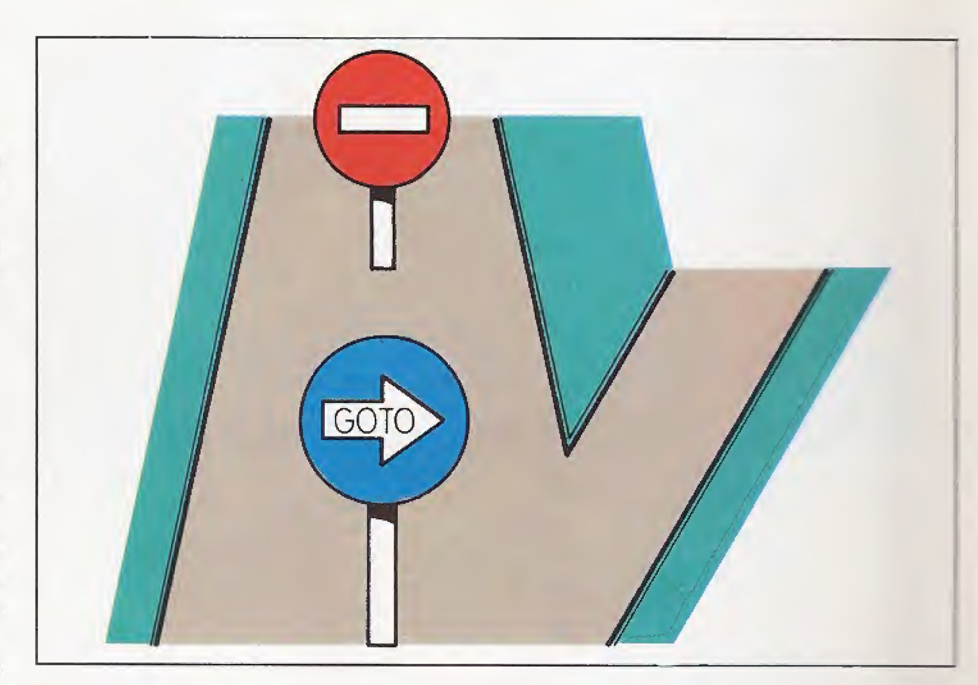
La sentencia IF sólo permite elegir una entre dos opciones: ejecutar la zona THEN o ejecutar la zona ELSE. Sin embargo, en muchos casos interesa poder elegir una entre varias opciones (más de dos). Para que ello sea posible utilizando sólo sentencias IF, será preciso construir estructuras artificiales, similares a la del último ejemplo.

Sin lugar a dudas ésta es una posibilidad de gran eficacia; pero queda aún otra sentencia de control, de tipo selectivo o de bifurcación condicional, que resolverá el problema con una mayor elegancia. Se trata de la sentencia CASE/OF, cuyo formato es el que se indica a continuación:

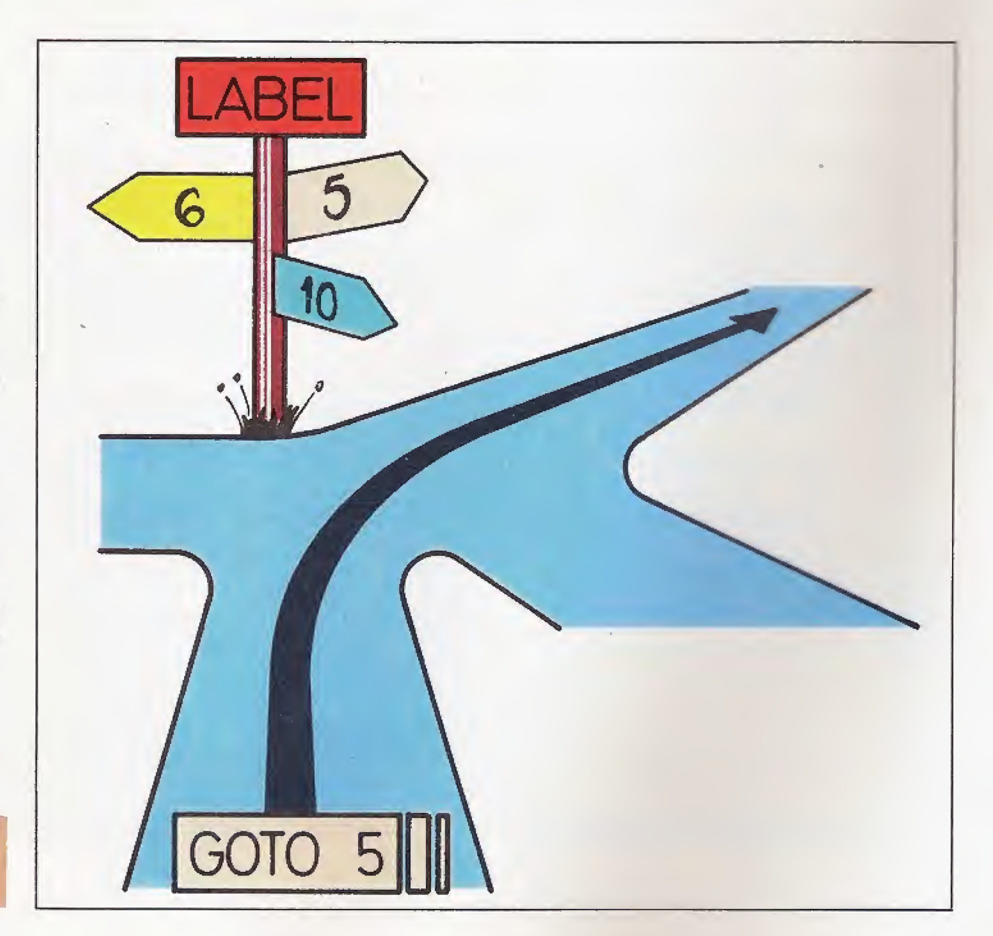
```
CASE <expr.> OF <et.>:<sent.>;
<et.>:<sent.>;...END;
```

Esta incluye una expresión, <expr.>, de tipo ordinal (INTEGER,CHAR,...), que seleccionará una única sentencia, simple o compuesta, de todas las emplazadas a continuación de la palabra OF y hasta la palabra END, que pone punto final a la sentencia CASE.

Cada una de las sentencias que forman el muestrario de opciones alternativas, debe ir precedida por una constante del mismo tipo que el de la expresión que actúa como selector. Estas constantes son denominadas «etiquetas CASE» (una de las pocas cosas que no hay de




*La sentencia GOTO ordena una bifurcación de tipo incondicional, rompiendo la secuencia normal de ejecución del programa.*




*Las etiquetas declaradas por medio de LABEL indican a la sentencia GOTO hacia qué punto del programa debe dirigir el flujo de ejecución.*




*El uso de GOTO resulta desaconsejable en PASCAL, puesto que rompe la estructura lógica del programa.*

declarar directamente en un programa PASCAL).

La estructura CASE seleccionará, en definitiva, aquella sentencia cuya etiqueta coincida con el valor actual que tome la expresión selectora, ignorando a las restantes. Una vez ejecutada la sentencia elegida, el flujo del programa abandonará la estructura CASE/OF.

Como es habitual, una etiqueta puede preceder a todo un conjunto de sentencias englobadas dentro de un bloque BEGIN/END. Si no existe una etiqueta coincidente con el valor obtenido por la expresión selectora, el efecto de la sentencia CASE es indefinido, produciendo normalmente un mensaje de error en la pantalla. Por ejemplo:

```
READLN(dia);
CASE dia OF
   1:nombre:='LUNES';
   2:nombre:='MARTES';
   3:nombre:='MIERCOLES';
   4:nombre:='JUEVES';
   5:nombre:='VIERNES';
   6:nombre:='SABADO';
   7:nombre:='DOMINGO';
END;
WRITELN(nombre);
```

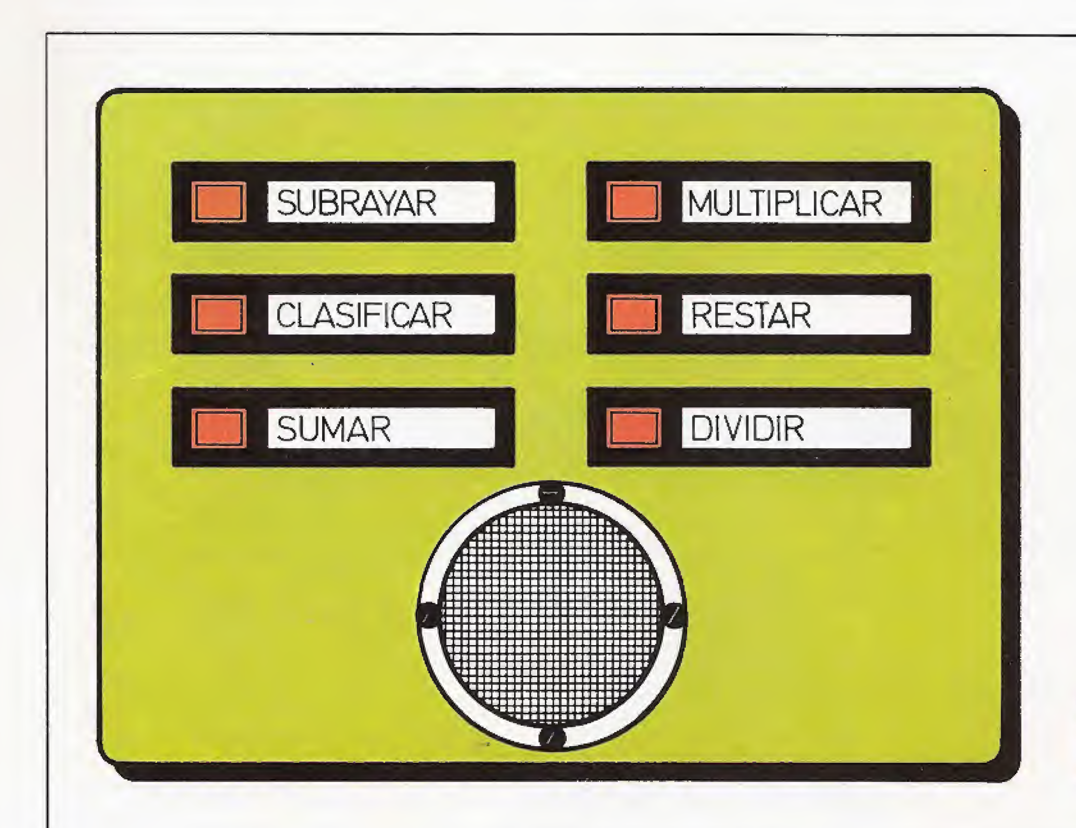
Como se ha indicado, si en el ejemplo anterior el valor de «dia» fuese menor que 1 o mayor que 7, el resultado sería indefinido. Para evitar este tipo de errores es conveniente limitar de alguna forma los valores que pueda tomar la expresión selectora. Ello puede lograrse de dos formas. La primera es incluir la sentencia CASE en una de las zonas de una sentencia IF:

```
IF dia>0 AND dia<8 THEN
   CASE dia OF...
ELSE WRITELN('CODIGO ERRONEO');
```

De esta forma, si «dia» toma un valor no comprendido entre 1 y 7, ambos incluidos, se imprimirá el mensaje que indica el error.

La segunda alternativa es declarar la variable selectora como de tipo subrango, con lo cual sólo podrá tomar determinados valores:

```
VAR dia: 1..7;
```




*La activación de un procedimiento se realiza mediante una simple llamada a su identificador.*

## Bifurcación incondicional

Para finalizar con las sentencias de control del PASCAL, queda por ver la de salto incondicional GOTO. Por medio de esta sentencia se puede dirigir el flujo del programa a cualquier punto del mismo, anterior o posterior a la posición que ocupe GOTO. Su formato general es muy simple; consta únicamente de la palabra clave GOTO, seguida por una etiqueta. Esta etiqueta, que será un número entero sin signo y, generalmente, de no más de cuatro cifras, se corresponderá con el valor que se sitúe inmediatamente delante de la sentencia a la que se desea transferir la ejecución del programa.

Cada etiqueta que vaya a ser utilizada con una sentencia GOTO, debe ser declarada previamente en la zona de declaraciones y definiciones del programa, mediante la palabra clave LABEL; ésta precederá a cualquier declaración CONST o VAR.

```
PROGRAM ejemplogoto (OUTPUT);
LABEL 100,200;
VAR n:INTEGER;

BEGIN
    n :=1 ;

100:IF n>10 THEN GOTO 200;
    WRITELN(n);
    n :=n+1;
    GOTO 100;
    200:END.
```

En el programa ejemplo se producirán bifurcaciones a las sentencias «etiquetadas» con los números 100 y 200 en diversos momentos de la ejecución.

El uso de sentencias GOTO rompe la estructuración del programa y, en consecuencia, dificulta su seguimiento y suele ser causa de múltiples errores. De ahí que su presencia sea desaconsejable en muchos casos; aunque hay veces que resulta imprescindible recurrir a ella, ya que su eliminación podría complicar el programa en exceso. En todo caso, una máxima fundamental de la programación estructurada aboga porque todas las sentencias GOTO pueden ser eliminadas de un programa.

«La presencia de GOTO indica que el programador no ha aprendido a pensar en PASCAL» (Jensen & Wirth).




*Los modernos lenguajes de programación —tal es el caso de PASCAL— tienden a ser estructurados y modulares. Ello facilita la confección y el mantenimiento de los programas.*

## Qué es un «procedimiento»

A la hora de confeccionar un programa, se presenta con frecuencia la necesidad de utilizar repetidamente, y en distintos puntos del mismo, un cierto bloque de instrucciones; instrucciones que realizan una tarea específica, coincidente en los distintos casos. Ello obliga a escribir el mismo conjunto de sentencias, tantas veces como sea necesaria su presencia a lo largo del programa. Este bloque de sentencias suele recibir el nombre de subprograma. En PASCAL es posible escribir cualquier subprograma una sola vez, y activarlo mediante una simple llamada al mismo cuando sea necesaria su ejecución.

Un programa es un edificio de compleja realización. Por ello, es importante que esté bien estructurado y sea fácil de leer; y no sólo por el traductor de lenguaje, sino también por el usuario o el programador. En base a ello, los modernos lenguajes de programación —tal es el caso del PASCAL—, tienden a ser estructurados y modulares. Sin lugar a dudas, el mantenimiento o modificación de un programa, será más sencillo si éste está dividido en pequeñas porciones (módulos) independientes.

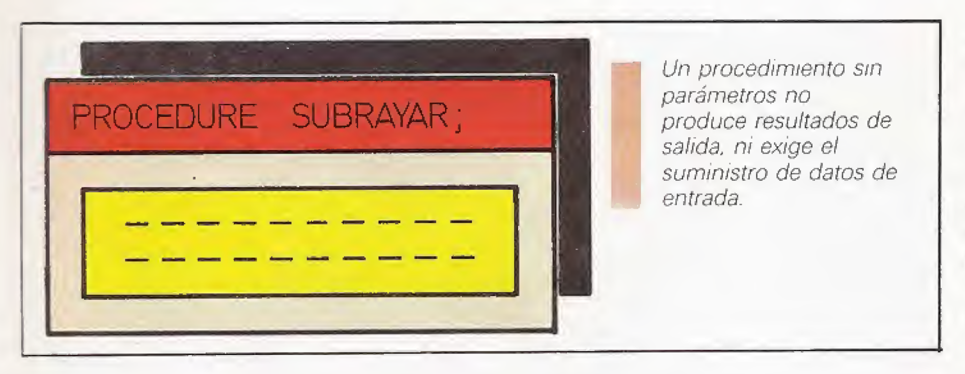
En PASCAL, un subprograma se denomina PROCEDURE (procedimiento). El conjunto de sentencias que lo constituyen se crean por medio de la «declaración del procedimiento». En ella se asocia un identificador al procedimiento, por medio del cual será activado en el instante apropiado.

La declaración de un procedimiento adopta la misma estructura que un programa PASCAL; razón por la que se puede considerar a un programa como un verdadero «superprocedimiento» PASCAL. En consecuencia, el procedimiento constará de una serie de parámetros de entrada/salida opcionales, de una zona de declaración de variables, también opcional, y finalmente de un cuerpo de sentencias englobado dentro de un bloque BEGIN/END. La declaración de un procedimiento debe ser posterior a la propia declaración de variables, por medio de VAR, del programa principal.

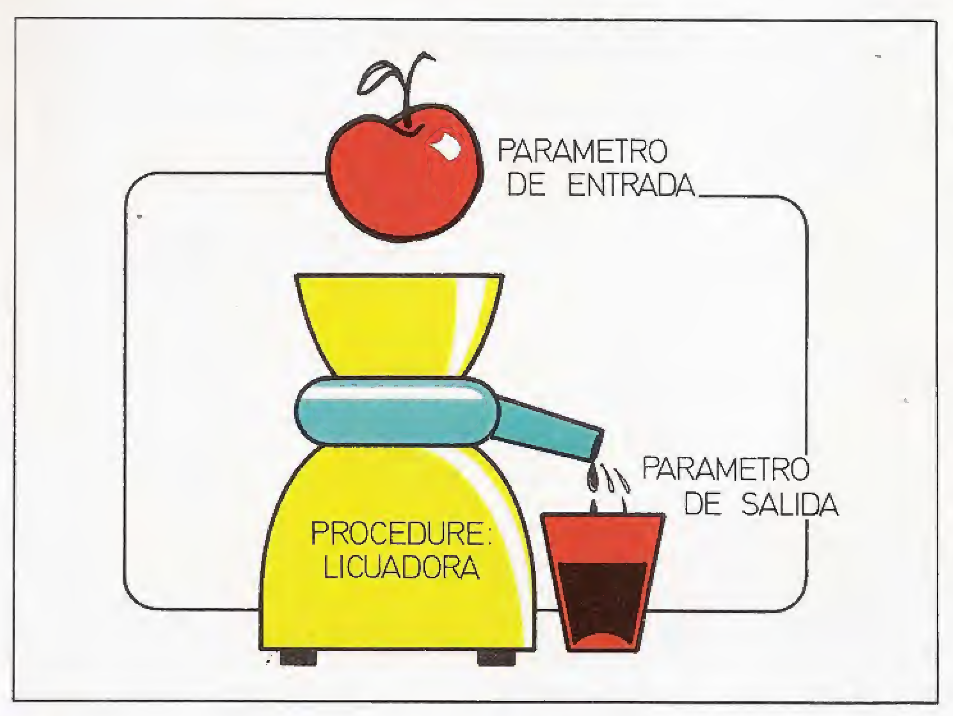
## Procedimientos sin parámetros

Como ya se mencionó, los parámetros de entrada/salida de un procedimiento son opcionales; es posible definir procedimientos sin parámetros y también procedimientos con parámetros.

Los procedimientos sin parámetros tienen como finalidad realizar una determinada acción en la que no se producirá ningún resultado aprovechable por otra zona del programa; a su vez, el procedimiento sin parámetros tampoco necesitará ningún dato elaborado por el




*Un procedimiento sin parámetros no produce resultados de salida, ni exige el suministro de datos de entrada.*




*Los procedimientos con parámetros pueden manipular datos externos a los mismos, así como crear y entregar datos a otras zonas del programa.*

resto del programa. El siguiente ejemplo ilustra la declaración de un procedimiento sin parámetros. Más adelante se analizarán con más detalle cada una de sus zonas constitutivas, así como su cometido.

```
PROCEDURE subrayar;
      VAR n:INTEGER;
      BEGIN
      FOR n:=1 TO final DO
          WRITE ('-');
      WRITELN
      END;
```

La estructura de la declaración es, tal y como se adelantó, totalmente similar a la de un programa. Tan sólo varía su cabecera: en lugar de utilizar la palabra clave PROGRAM, se emplea ahora la palabra PROCEDURE.

La palabra que identifica a este procedimiento es «subrayar», y debe ser un identificador válido en PASCAL. Su cometido será activar el procedimiento en el punto adecuado del programa, además de dar una idea del objetivo del procedimiento; esto último siempre que el nombre utilizado resulte sugerente. El procedimiento del ejemplo tiene como misión visualizar en el dispositivo de salida (la pantalla), una línea de guiones, que actuará como subrayado de la línea que se haya imprimido con anterioridad.

La zona de declaraciones de un procedimiento es totalmente análoga a la de un programa. De nuevo, su misión es la de declarar las variables que vayan a ser utilizadas únicamente dentro del procedimiento. En el caso de la variable «final», su valor le será asignado fuera del procedimiento y, por lo tanto, no es necesaria su declaración dentro del mismo, aunque sí está permitido el referirse a ella. La variable «n» se dice que es una variable local, que sólo podrá ser referida dentro de ese procedimiento, mientras que la variable «final» es una variable global, que puede ser utilizada en cualquier zona del programa. Por lo tanto, hay que tener en cuenta que un mismo identificador, declarado como variable global a nivel de programa y a la vez como variable local en un procedimiento, no representará a la misma variable, recibiendo valores diferentes en cada zona del programa.

Finalmente, el cuerpo de sentencias encerrado entre las palabras BEGIN y END incluirá el conjunto de sentencias ejecutables que se encargarán de realizar la misión encomendada al procedimiento. En principio, puede parecer un contrasentido el hecho de incluir sentencias ejecutables en la zona de declaraciones del programa; ello se puede interpretar como una indicación de la acción a realizar al ser activado el procedimiento.

Para activar un procedimiento sin parámetros basta tan sólo con escribir la palabra o identificador que se colocó detrás de la palabra PROCEDURE. El siguiente programa utiliza el procedimiento creado en el ejemplo anterior:

```
PROGRAM cabecera (INPUT,OUTPUT);
VAR pag,final:INTEGER;
PROCEDURE subrayar;
      VAR n:INTEGER;
      BEGIN
            FOR n:=1 TO final Do
            WRITE('—');
            WRITELN
      END;
BEGIN
      WRITELN('HASTA QUE COLUMNA');
      READLN(final);
```

**TABLA DE COMANDOS-PASCAL**

| Comando | Cometido | Observaciones |
|---|---|---|
| WHILE <expr.> DO <sent.>; | Ejecución de un bucle mientras es cierta la expresión | Cuerpo del programa |
| REPEAT <sent.> ;<sent.>; ... UNTIL <expr.>; | Ejecución de un bucle hasta que sea cierta la expresión | Cuerpo del programa |
| FOR <cont.>:=<ini.> TO <fin.> DO <sent.>; | Ejecución de un bucle según los valores ascendentes de una variable | Cuerpo del programa |
| FOR <cont.>:=<ini.>DOWNTO<fin.>DO<sent.>; | Ejecución de un bucle según los valores descendentes de una variable | Cuerpo del programa |
| BEGIN <sent.> ;<sent.>; ... END; | Agrupación de varias sentencias en una única compuesta | Cuerpo del programa |
| IF <expr.> THEN <sent.> {ELSE <sent.>}; | Ejecución condicional de una entre dos opciones | Cuerpo del programa |
| CASE <sel.> OF <et.>:<sent.>;...END; | Ejecución condicional de una entre dos o más opciones | Cuerpo del programa |
| GOTO <etiqueta>; | Bifurcación incondicional | Cuerpo del programa |
| LABEL <etiqueta>; | Declaración de etiquetas | Zona de definiciones y declaraciones |
| PROCEDURE <id.>; <declaraciones>; <bloque de sentencias>; | Declaración de un subprograma o procedimiento sin parámetros | Zona de declaraciones y definiciones. |
| PROCEDURE <id.>; (<lista>); <declaraciones>; <bloque de sentencias>; | Declaración de un subprograma o procedimiento con parámetros | Zona de declaraciones y definiciones |
| <id.>{, <id.>, ...}: <tipo>; | Parámetros formales de entrada | Cabecera de la declaración de procedimientos con parámetros |
| VAR <id.>{,<id.>, ...}: <tipo>; | Parámetros formales de entrada/salida | Cabecera de la declaración de procedimientos con parámetros |

NOTA: <expr.>: expresión de tipo booleano. <sent.>: sentencia ejecutable. <cont.>: variable de control del bucle. <ini.>: expresión que da el valor inicial a la variable de control. <fin.>: expresión que indica el valor final de la variable de control. <sel.>: expresión de tipo ordinal. <et.>: etiqueta; será una constante del mismo tipo que el selector. <etiqueta>: entero de no más de cuatro cifras y sin signo. <id.>: identificador válido PASCAL. <tipo>: tipo de datos. <lista>: lista de parámetros formales de entrada y/o salida de un procedimiento.

```
    WRITELN('SUBRAYADO HASTA LA
    COLUMNA:',final)
    subrayar
END.
```

La sentencia de llamada que activa al procedimiento es su identificador «subrayar», tal y como se observa en el ejemplo. Ello desencadena la ejecución de las acciones indicadas en el bloque de sentencias. Si la ejecución desea repetirse en cualquier otra zona del programa, bastará con hacer una nueva llamada al procedimiento en cuestión utilizando su identificador: «subrayar».

## Procedimientos con parámetros

Cuando se desee que un determinado subprograma pueda manejar datos creados en otra zona del programa, o incluso crear sus propios datos para utilizarlos en el resto del programa, habrá que recurrir a los denominados procedimientos con parámetros. El término «parámetros» se refiere a esos datos que manipula el procedimiento y que tienen una procedencia o destino exterior al procedimiento. En consecuencia, los parámetros pueden ser de entrada, de salida o de entrada y salida a la vez. Estos parámetros se indicarán encerrados entre paréntesis, detrás del identificador del procedimiento; algo semejante a lo que se hace en la cabecera del programa. Esta es la única diferencia sustancial respecto a los procedimientos sin parámetros. La forma más general de un procedimiento con parámetros es la siguiente:

```
PROCEDURE <identificador> (<param.> [;<param.>;...]);
<zona de declaraciones>;
<bloque de sentencias>.
```

Los parámetros reflejados en la cabecera del procedimiento se llaman *parámetros formales,* debido a que en el momento de declarar el procedimiento no tienen asignado ningún valor concreto. El valor se asignará al hacer la llamada al procedimiento, tal y como analizaremos más adelante. Los parámetros formales de entrada del procedimiento adoptarán la forma siguiente:

```
<identificador>:<tipo>;
```

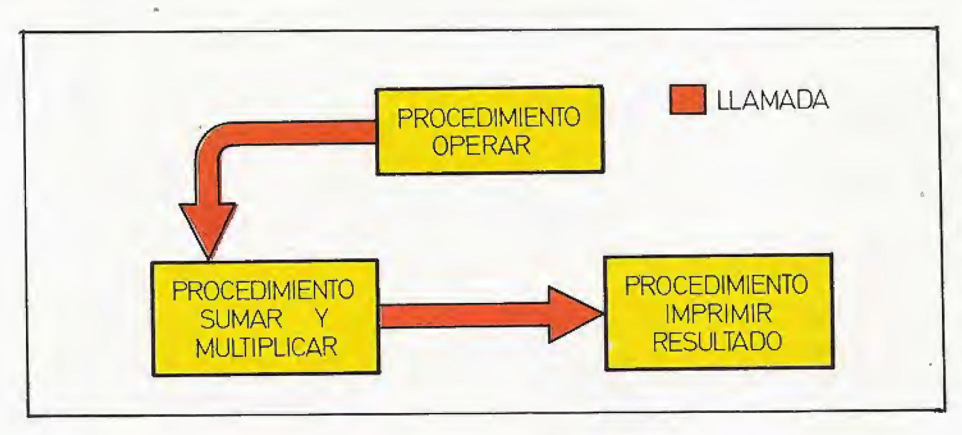
En este caso, al ser tan sólo un parámetro de entrada, su valor no estará disponible fuera del procedimiento; esto es: no podrá coincidir con un resultado del mismo. En cambio, los parámetros formales de entrada y salida, adoptarán un nuevo aspecto:

```
VAR <identificador>:<tipo>;
```

Ahora, se obtendrá en este parámetro




*Los parámetros actuales de la sentencia de llamada transfieren los datos a los parámetros formales del procedimiento.*




*Un procedimiento puede activar a otro distinto y éste, a su vez, a un tercero; y así sucesivamente, creando estructuras más complejas.*

un resultado del procedimiento que puede ser utilizado en cualquier otra zona del programa. Un ejemplo contribuirá a clarificar las cosas:

```
PROGRAM operaciones (INPUT,OUTPUT);
VAR A,B,suma,producto:REAL;
PROCEDURE sumar (num1,num2:REAL;
VAR resulsuma:REAL);
BEGIN
    resulsuma:=num1+num2
END;
PROCEDURE multiplicar (num3,num4:
  REAL;VAR resulproducto:REAL);
BEGIN
    resulproducto:=num3*num4
END;
BEGIN (*del programa principal*)
    READLN(A,B);
    sumar(A,B,suma);
    multiplicar(A,B,producto);
    WRITELN('SUMA=',suma);
    WRITELN('PRODUCTO=',producto)
END (* del programa *).
```

En el programa intervienen dos procedimientos con parámetros, denominados «sumar» y «multiplicar». En ellos, los parámetros formales son: «num1», «num2», «num3» y «num4» (sólo de entrada) y «resulsuma» y «resulproducto» (de salida).

La llamada a los procedimientos con parámetros se realiza en la forma habitual, por medio de su identificador; si bien, en este caso, hay que añadir tras él una lista de parámetros encerrada entre paréntesis.

Estos parámetros, incluidos en la sentencia de llamada, se denominan *parámetros actuales,* por ser los que van a asignar valores a los parámetros formales del procedimiento. En el ejemplo, al parámetro formal «num1» se le asignará el valor del parámetro actual «A» en el momento de la llamada; y análogamente para los restantes parámetros de entrada al procedimiento. El procedimiento operará dichos valores para producir un resultado que se asignará al correspondiente parámetro de salida; su valor estará disponible en el parámetro actual de salida de la sentencia de llamada («suma» o «producto»), una vez concluida la ejecución del procedimiento. Cada parámetro actual inicializa a su correspondiente parámetro formal; por ello, es necesario que éstos se correspondan en número, orden y tipo, en la sentencia de llamada y en la declaración del procedimiento.

Suponga que al ejecutar el programa se introducen para «A» y «B» los valores 3 y 5, respectivamente. En el momento de llamar al procedimiento «sumar», el valor «A» se transfiere a «num1» y el de «B» a «num2». El procedimiento asignará entonces el valor 8 a «resulsuma», y al devolver el control a la sentencia de llamada, este valor será transferido a «suma», con lo que este resultado estará disponible en dicho parámetro formal es también de salida, su parámetro actual para imprimirlo en la pantalla o en el periférico de salida.

Si un parámetro formal es sólo de entrada, el correspondiente parámetro actual puede ser una constante, una expresión, una variable, o incluso el identificador de llamada a otro procedimiento. No obstante, si el parámetro formal es también de salida, su parámetro actual debe ser obligatoriamente una variable. Por ejemplo:

```
sumar(3,5+B,suma);
multiplicar(3,5,9);
```

Así la llamada anterior al procedimiento «sumar» del programa «operaciones», es correcta, pero no lo es la llamada a «multiplicar», puesto que el parámetro actual de salida es una constante (9) en lugar de ser una variable.

# PASCAL (3)

## Funciones y tipos de datos no estructurados

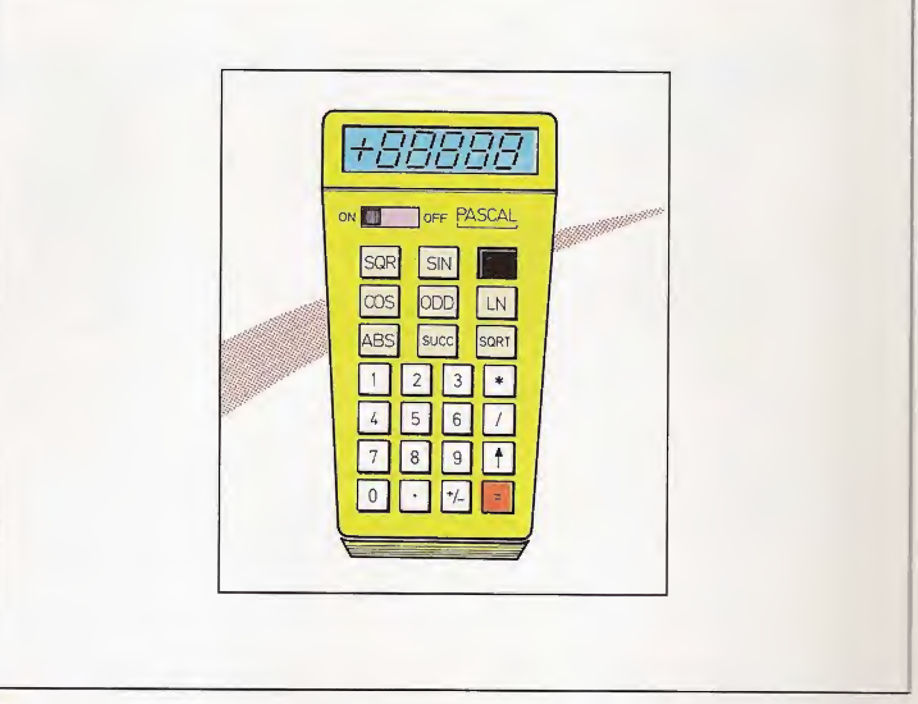


La modularidad de que gozan los programas codificados en lenguaje PASCAL, no se limita única y exclusivamente a los procedimientos. Hay otros medios que permiten independizar a una determinada zona del programa y englobarla en lo que se denomina un «subprograma». Este subprograma, encargado de realizar una función específica, podrá utilizarse con total comodidad en cualquier otro punto del programa; y sin que por ello haya que escribir de nuevo el mismo conjunto de sentencias cada vez que se desee su intervención.

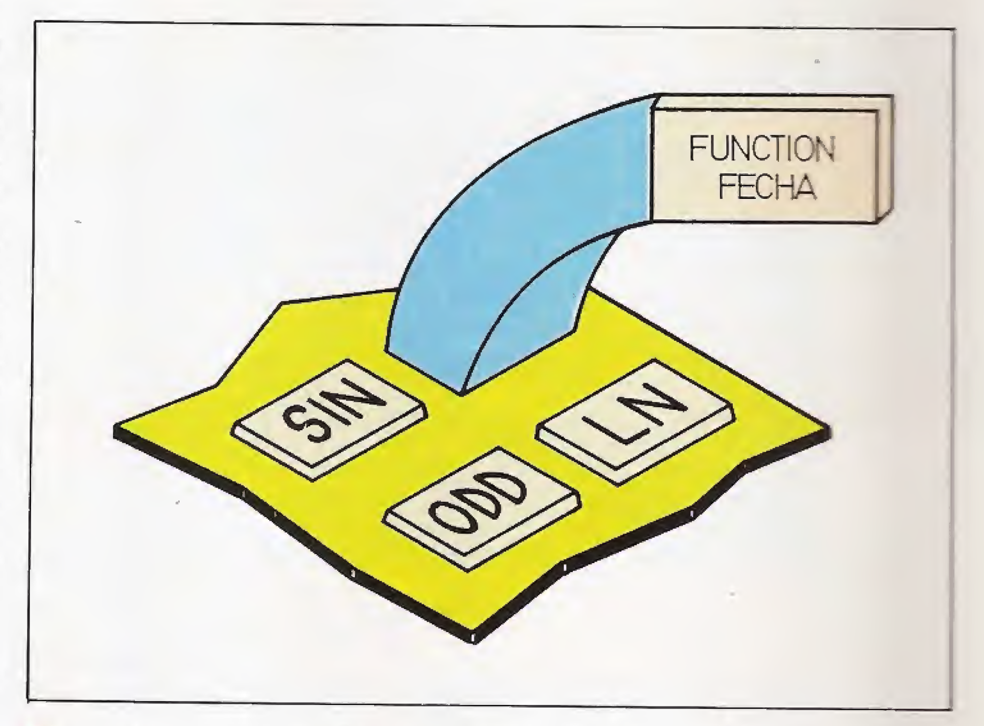
Para que un bloque de sentencias adquiera la condición de subprograma, no es indispensable que su ejecución deba repetirse en diversos puntos del programa. Aunque su ejecución deba realizarse tan sólo una vez, es recomendable segmentar cuidadosamente el programa en subprogramas que refuercen su estructura modular. Con ello, el programa resultará más inteligible para cualquier persona que lo lea y, a su vez, será más fácil y rápida su depuración y modificación.

Habitualmente es preciso manipular una determinada información, transformándola según un criterio específico. Suponga, por ejemplo, que se tiene como información de partida los datos BASE y ALTURA (dimensión de la base y altura) de un triángulo. Esta información puede ser manipulada en orden a obtener como resultado el área del triángulo en cuestión. El pequeño subprograma al que se encomiende la función de hallar el área del triángulo, sólo tendrá que tomar los datos de entrada (BASE y ALTURA), obtener su producto, tras ello dividirlo por dos y, por último, asignar este valor resultante a otra variable, por ejemplo: AREA. El referido tratamiento de los datos se podría realizar con la ayuda de un procedimiento con parámetros de entrada y de salida.

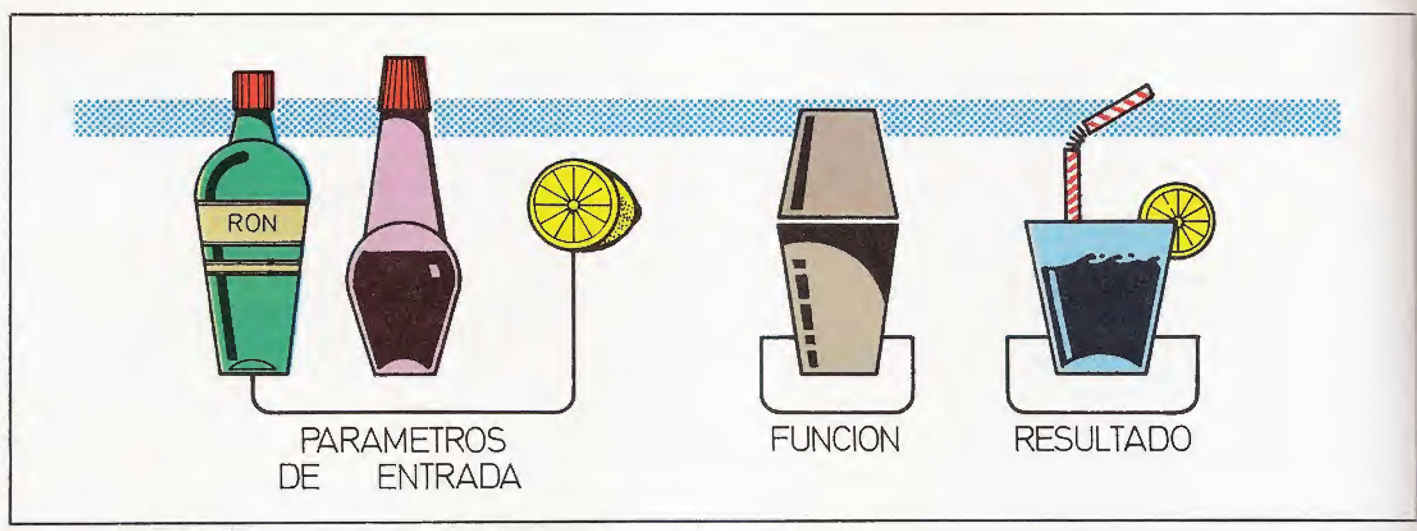
```
PROCEDURE areatriángulo (BASE,ALTURA:REAL;VAR
AREA:REAL);
BEGIN
  AREA:=BASE*ALTURA/2
END;
```




*Dentro del repertorio del lenguaje PASCAL existen toda una serie de funciones ya definidas y a disposición directa del usuario.*




*El usuario puede completar la gama de funciones PASCAL creando, a voluntad, sus propias funciones; para ello debe utilizar la palabra clave FUNCTION.*




*Una función con parámetros formales de entrada obtiene un resultado dependiente del valor actual de los referidos parámetros.*

Existe, como ya se apuntó, otra posibilidad para confeccionar este segmento de programa. Esta no es otra que la de definir el referido conjunto de sentencias como una «función», capaz de generar un resultado.

## Las funciones del PASCAL

Una función consiste en una serie de órdenes de cálculo, aplicadas sobre ciertos operandos de entrada, que proporcionan un resultado de salida.

Las funciones deben ser declaradas en la correspondiente zona de declaraciones y definiciones del programa previamente a su uso, al igual que ocurría en el caso de los procedimientos.

La declaración de las funciones se hace de forma similar a la declaración de los procedimientos; si bien, ahora es preciso utilizar la palabra reservada FUNCTION (función) en la cabecera de la declaración. Esta irá seguida por un identificador válido PASCAL, que se utilizará posteriormente para llamar a la función, y por una lista opcional de parámetros formales de entrada. Los parámetros aparecerán, con sus nombres y sus tipos, encerrados entre paréntesis. Por último, tras la lista de parámetros se especificará el tipo de resultado que proporciona la función.

A esta cabecera le seguirá la zona de declaración de variables, en el caso de ser necesario, y el cuerpo de la función. En ella es realmente donde se calcula el resultado a partir de los parámetros actuales de entrada. En definitiva, el aspecto general de una función es el siguiente:

```
FUNCTION <identificador> [(<parámetro formal> [,...])]:
<tipo de resultado>;
<zona de declaraciones>;
<cuerpo de la función>;
```

*En el ejemplo del texto, la función AREA calcula el valor de la superficie de un triángulo a partir de dos parámetros de entrada: BASE y ALTURA.*

Comúnmente suele establecerse una división entre las funciones, separándolas en funciones sin parámetros y funciones con parámetros. En realidad, se puede considerar que la lista de parámetros es opcional y, por lo tanto, es facultativo incluirla cuando se considere necesaria u omitir su presencia.

Cabe precisar que una función sólo puede calcular un único valor como resultado, a diferencia de los procedimientos con parámetros que pueden generar más de un resultado de salida.

El resultado debe ser asignado al identificador de la función, dentro del cuerpo de sentencias que la componen. Será esta asignación la que permitirá «devolver» el resultado de la función. Al respecto, resulta obvio que el tipo de valor calculado debe coincidir con el tipo del dato declarado en la cabecera como resultado de la función. Por lo demás, el resultado sólo puede ser de un tipo escalar (REAL, INTEGER, CHAR o BOOLEAN); no es admisible que el resultado de una función sea de tipo estructurado.

El lenguaje PASCAL cuenta con una serie de funciones estandarizadas, cuyo empleo por parte del usuario no exige su declaración previa. Ejemplos de funciones estándar incorporadas al PASCAL y que ya se han mencionado en anteriores capítulos de la obra son: SQR(X), SIN(X), ORD(X), SUCC(X), etc.

Volviendo al ejemplo precedente (cálculo del área de un triángulo), podemos codificarlo ahora por medio de una función:

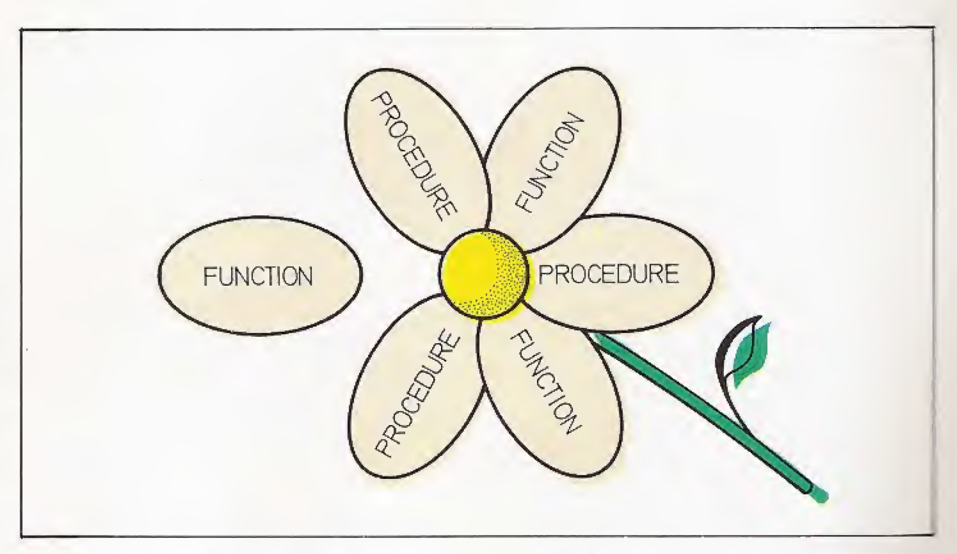
```
PROGRAM AREATRIANGULO (INPUT, OUTPUT);
VAR longbase,longaltura :REAL;
FUNCTION area (BASE,ALTURA: REAL):
  REAL;
BEGIN
area:=BASE*ALTURA/2
END;
BEGIN(*del programa principal*)
  READLN(longbase,longaltura);
  WRITELN('AREA=',area(longbase, longaltura)
END.
```

Aquí se observa con total claridad cómo se declara una función para que entregue un resultado de tipo real (REAL).

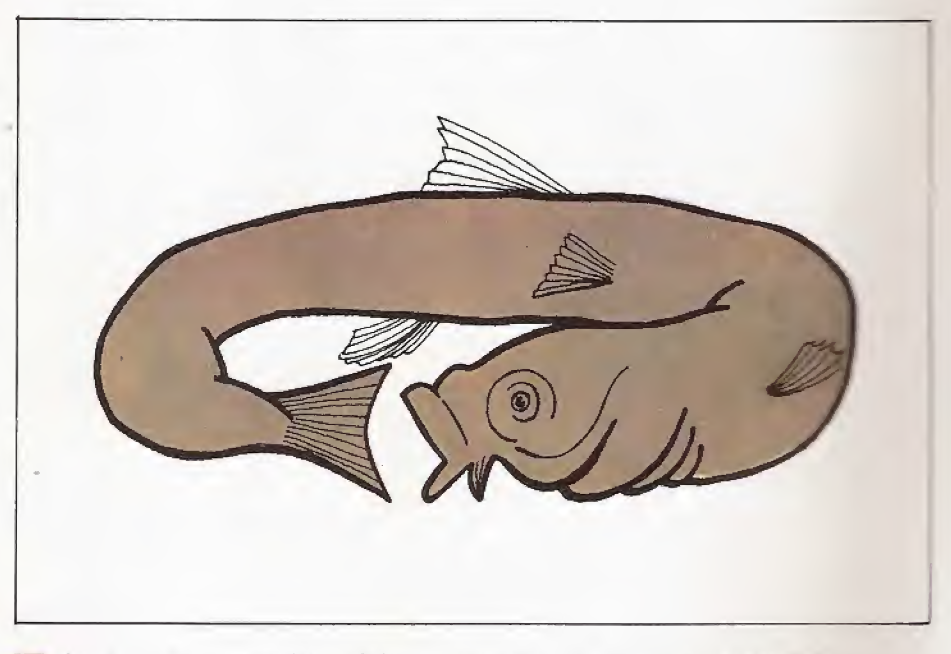
En el cuerpo de la función, que debe ir escoltado por las palabras BEGIN y END, se asigna el valor calculado al identificador «area». Tal asignación permitirá que la función «devuelva» el valor calculado al punto de llamada.

La llamada o activación de una función PASCAL, se puede realizar de dos formas. La primera se concreta en una llamada directa, sin más que nombrar en el momento adecuado el identificador de la función, seguido por la lista de parámetros actuales si los hubiere. Así es como tiene lugar en el ejemplo anterior: la llamada se realiza dentro de una sentencia de escritura que presentará directamente el resultado de la función. En el segundo caso, la llamada se realiza dentro de una sentencia de asignación o de comparación, por ejemplo:

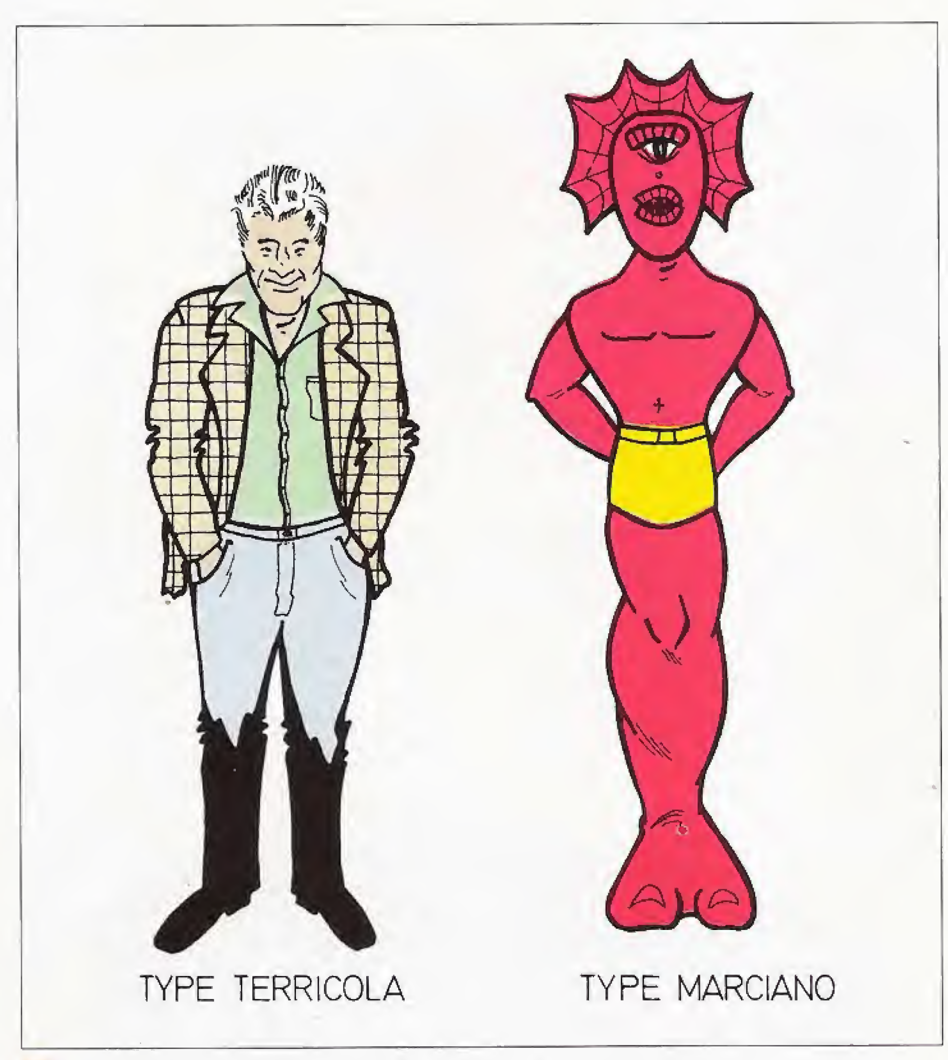
```
AREADELTRIANGULO:=area(BASE,ALTURA);
```




*En muchos casos es difícil decidir cuál es el método idóneo para resolver un determinado problema... ¿Procedimiento, o función?*



*Las estructuras recursivas reflejan en el ámbito informático la imagen de la pescadilla que se muerde la cola: ¿se están alimentando, o sirven de alimento?*




*La posibilidad de definir distintos tipos de datos permite diferenciar a éstos entre sí. La propia definición de tipo delimita, al tiempo, los posibles valores que puede adoptar cada dato.*

## «Function» versus «procedure»

Cabe pensar que cualquier segmento de programa codificable a modo de procedimiento, puede codificarse análogamente mediante una función. Tal afirmación es cierta, debido a la gran similitud que existe entre ambos conceptos. De ahí que quepa la duda de cuándo utilizar un procedimiento (PROCEDURE) y cuándo una función (FUNCTION).

En la práctica existen claras diferencias entre procedimiento y función. De entrada, una función puede generar un único resultado, mientras que un procedimiento puede entregar cualquier número de valores de salida (uno, varios o ninguno). Por otro lado, los parámetros formales de una función sólo son de entrada —aportan valores a la función—, mientras que los parámetros de un procedimiento pueden ser de entrada y salida de datos.

Para precisar el uso de uno u otro método en cada caso específico, es conveniente recordar la siguiente regla. Cuando sólo sea necesario un único resultado, teniendo o no parámetros de entrada, se debe utilizar una FUNCTION. En los demás casos, esto es, cuando no haya resultado o haya más de uno, se utilizará el PROCEDURE. Por supuesto que esta regla no es inquebrantable, pero su aplicación facilitará, sin lugar a dudas, un uso más eficaz de los recursos que brinda el lenguaje PASCAL.

## La recursividad del PASCAL

La recursividad se define como la capacidad de un subprograma para llamarse a sí mismo. En efecto, dentro de un procedimiento o función se puede incluir una llamada a ese mismo procedimiento o función. Para clarificar este concepto, puede examinarse la función siguiente, que calculará de forma recursiva el factorial de un número entero. (El factorial de un número se define como el producto de todos los enteros comprendidos entre 1 y dicho número, ambos inclusive).

```
FUNCTION factorial (N:INTERGER):
  INTERGER;
BEGIN
  IF N=1 THEN factorial:=1
    ELSE factorial:=factorial(N-1)*N
END;
```

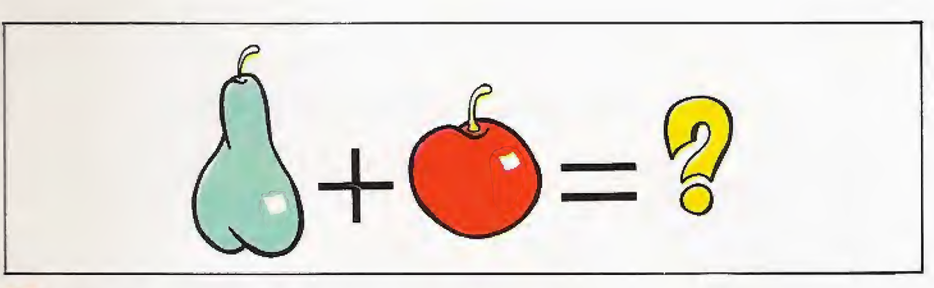
En el ejemplo se observa cómo dentro del cuerpo de la función «factorial» se efectúa una nueva llamada a la misma función «factorial», aunque ahora con otro parámetro actual de entrada. Suponga que en un determinado programa se realiza la siguiente llamada a esta función:

```
NUMERO:=factorial(3);
```

al intentar obtener el resultado para el valor actual (3) del parámetro formal «N», la función detectará que dicho valor es distinto de 1 (IF N=1 ...). A raíz de ello, se producirá una nueva llamada a la función ahora con el valor de entrada 2, dejando pendiente el producto de 3 por el resultado de la función «factorial (2)». La nueva llamada se encontrará con una situación parecida a la del caso anterior, operándose una nueva llamada a la función factorial, esta vez con el parámetro actual 1. De nuevo, quedará pendiente el producto del 2 por el resul-

tado de la función «factorial (1)». La llamada «factorial (1)» cerrará este encadenamiento de llamadas, devolviendo ahora el valor 1 al ser cierta la condición impuesta en la sentencia IF, con lo que se podrán realizar todas las operaciones pendientes de las llamadas anteriores.

Dentro de una estructura recursiva debe figurar siempre alguna sentencia de ruptura; de lo contrario se formaría un ciclo sin final. En él se repetiría incesantemente la llamada a la función o al procedimiento, hasta que se originara un error por desbordamiento (overflow) de alguna de las variables implicadas en la estructura recursiva.




*Mediante la introducción de tipos de datos definidos por el usuario puede evitarse el tradicional error de operar con datos de distinto tipo... ¡sumar peras con manzanas!*

## Los tipos del PASCAL

Para que sea posible utilizar cualquier elemento del PASCAL, ya se trate de constantes, variables, funciones o expresiones, es imprescindible su declaración previa dentro del programa. Esta declaración previa permite asignar a cada elemento del PASCAL lo que denominamos un «tipo»; de forma que cada elemento será de un tipo determinado y sólo uno. El tipo de cada dato determinará el conjunto de posibles valores que éste puede adoptar, así como los operadores que se podrán utilizar con él. No hay que olvidar que cada tipo de datos admite un determinado conjunto de operadores que producirán resultados de ese mismo tipo.

El hecho de distinguir entre tipos de datos facilita la tarea de programación y agiliza la detección de errores. Al trabajar con lenguajes que no utilizan tipos de datos, es frecuente que se produzcan errores muy difíciles de detectar; por ejemplo, al intentar «sumar peras con manzanas» (cantidades ambas que estarán expresadas en forma de números enteros). Al utilizar diferentes tipos de datos para las peras y las manzanas, quedan eliminados los posibles errores de esta naturaleza.

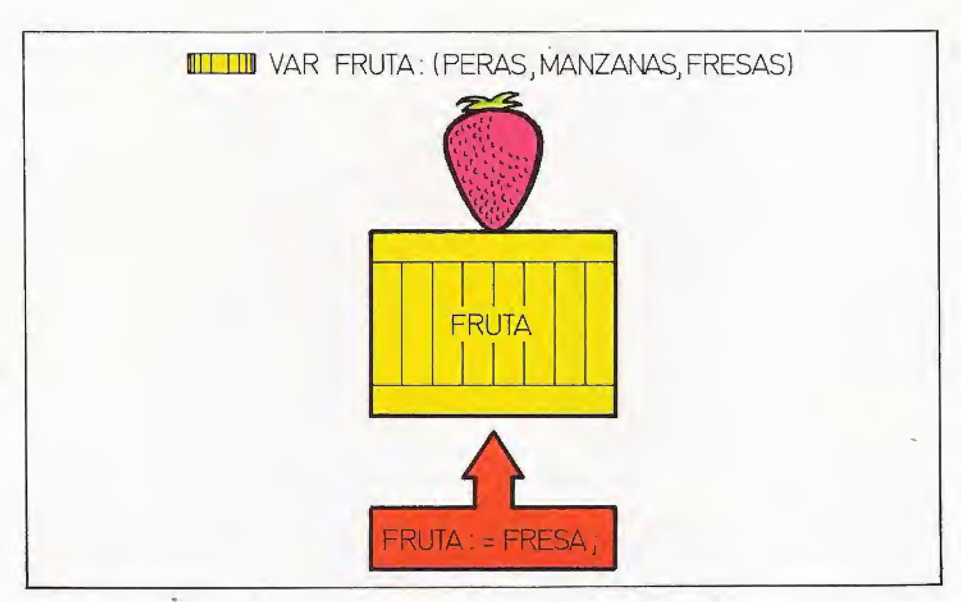
En el capítulo precedente se vio cómo la declaración de las variables con VAR asigna instantáneamente el tipo especificado a dichas variables. Asimismo, mediante la declaración de constantes con CONST, era posible que una futura modificación de esos valores constantes sólo afectara a su declaración, y no a todas las sentencias en las que figurase esa constante. De forma similar, el PASCAL permite dar un nombre o identificador a un tipo de dato determinado mediante la llamada «definición de tipo». Ello contribuye a facilitar en mayor medida el manejo de los tipos de datos. El formato de esta definición de tipo es el siguiente:

```
TYPE <identificador> = <tipo>;
```

Toda definición de tipo debe preceder en el programa a la declaración de variables con VAR. Es importante reseñar que esta definición de tipo NO declara a una variable, sino únicamente a un tipo de dato con un nuevo nombre; nombre que se podrá utilizar posteriormente como tal en la declaración de variables




*La definición de datos del tipo enumerado se lleva a efecto relacionando ordenadamente cada uno de los nuevos valores a crear.*




*La declaración de una variable como de tipo enumerado puede también realizarse mediante la palabra clave VAR.*

o de otros elementos del PASCAL. En el siguiente ejemplo se observa la puesta en práctica de una definición de esta índole:

```
TYPE
    numero=REAL;
    respuesta=BOOLEAN;

VAR
    temperatura:numero;
    calor:respuesta;
```

Estas declaraciones equivalen a las siguientes:

```
VAR
    temperatura:REAL;
    calor:BOOLEAN;
```

## Tipos de datos no estructurados

Tipos de datos no estructurados, o escalares estándar, son aquellos cuyos posibles valores no se pueden descomponer en otro tipo de dato. Estos tipos de escalares estándar, ya estudiados en anteriores capítulos de la obra, son: INTEGER, REAL, CHAR y BOOLEAN.

Los posibles valores de estos tipos de datos están ordenados, esto es: al evaluar dos datos, siempre se puede establecer un orden ascendente o descendente entre ellos. En virtud de esta característica, quedan definidos los operadores de relación (menor que, mayor o igual que...), así como la sentencia de asignación. Una expresión de cualquier tipo escalar, excepto el REAL, puede ser variable de control de un bucle FOR o selector de una sentencia CASE/OF.

No obstante, hay muchas ocasiones en las que el usuario desearía poseer otros tipos de datos, que le permitieran manejar con mayor facilidad una cierta información. Así, por ejemplo, en determinadas ocasiones, convendría utilizar un tipo de datos cuyos valores fuesen los nombres de los días de la semana, o los tres colores básicos; o incluso los palos de una baraja de cartas española. El PASCAL permite la definición de nuevos tipos de datos escalares por parte del usuario. Esta definición puede llevarse a efecto según dos métodos: por enumeración, o por definición de un subconjunto de cualquier otro tipo de dato escalar (tipos subrango).

## El tipo enumerado

La definición de tipo por enumeración se realiza especificando ordenadamente todos y cada uno de los valores del nuevo tipo que se quiere definir. Los valores se representarán por medio de identificadores; éstos constituirán las constantes de ese tipo de datos definido, y se declararán mediante la siguiente definición:

```
TYPE <identificador del tipo>=(<id.1>,<id.2>,...,<id.n>);
```

El identificador del tipo será el nuevo nombre de ese tipo de datos, mientras que la lista encerrada entre paréntesis coincidirá con los posibles valores que pueden adoptar los elementos que se declaren. Veamos un ejemplo al respecto:

```
TYPE FRUTAS:(PERAS,MANZANAS,FRESAS);

VAR f:FRUTAS;
```

En él, la sentencia de asignación «f:=FRESAS» es válida y plenamente correcta, puesto que la variable «f» ha sido declarada del tipo FRUTAS, entre cuyos posibles valores se encuentra el de FRESAS. Sin embargo, no será posible asignar a «f», o a cualquier otra variable declarada con el tipo FRUTAS, otro valor distinto de los nombres PERAS, MANZANAS o FRESAS; en ese caso se produciría un error.

La definición de los tipos por enumeración se puede efectuar también directamente de la siguiente forma:

```
VAR f:(PERAS,MANZANAS,FRESAS);
```

Un tipo de datos definido por enumeración y que ya está implementado en el lenguaje PASCAL, es el tipo BOOLEAN. Las constantes de este tipo de datos son TRUE y FALSE. Virtualmente, es como si el programador hubiese realizado la siguiente definición:

```
TYPE BOOLEAN=(TRUE,FALSE)
```

No obstante, y como quiera que están ya disponibles directamente, el programador no se verá obligado a repetir dicha definición cada vez que desee utilizar el tipo BOOLEAN.

Los identificadores empleados como constantes de un nuevo tipo enumerativo de datos recién definido, no pueden volver a utilizarse como constantes den la definición de otro nuevo tipo de da-

tos. En consecuencia, no será válida la siguiente definición de tipos, pues el tipo al que pertenece el valor SAB queda definido de forma ambigua, al no ser de un tipo único:

```
TYPE
  CURRAR (LUN,MAR,MIE,JUE,VIE,SAB);
  DESCANSAR: (SAB,DOM);
```

Los tipos definidos por el usuario serán de uso exclusivamente interno del programa. Desde luego, pueden ser asignados o probados; si bien, no se podrán utilizar en sentencias de lectura o escritura, es decir, no se podrán asignar mediante sentencias READ o READLN, ni esciribir sus valores con WRITE o WRITELN.

## Tipos subrango

Para definir sus propios tipos de datos, el programador cuenta con un segundo método: la definición de los llamados tipos subrango. Este nuevo tipo de datos, que puede ser creado a partir de cualquier otro tipo de dato escalar definido con anterioridad (excepto del tipo REAL), consiste en un subconjunto de valores de ese otro tipo, limitado por un valor máximo y un valor mínimo. Su definición adopta el siguiente aspecto:

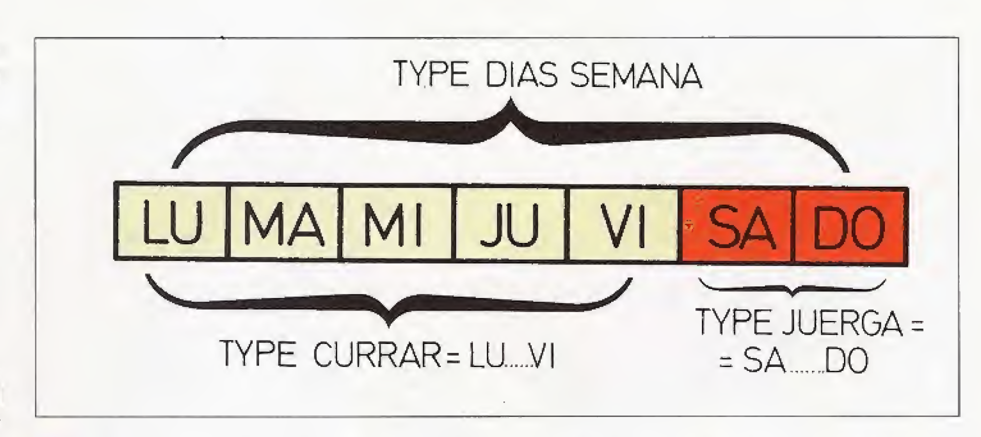
```
TYPE <identificador tipo>=<cte.>...
<cte.>;
```

Las dos constantes que delimitan el rango de valores permisible deben ser del mismo tipo, y además deben estar ordenadas: límite inferior a la derecha del límite superior.

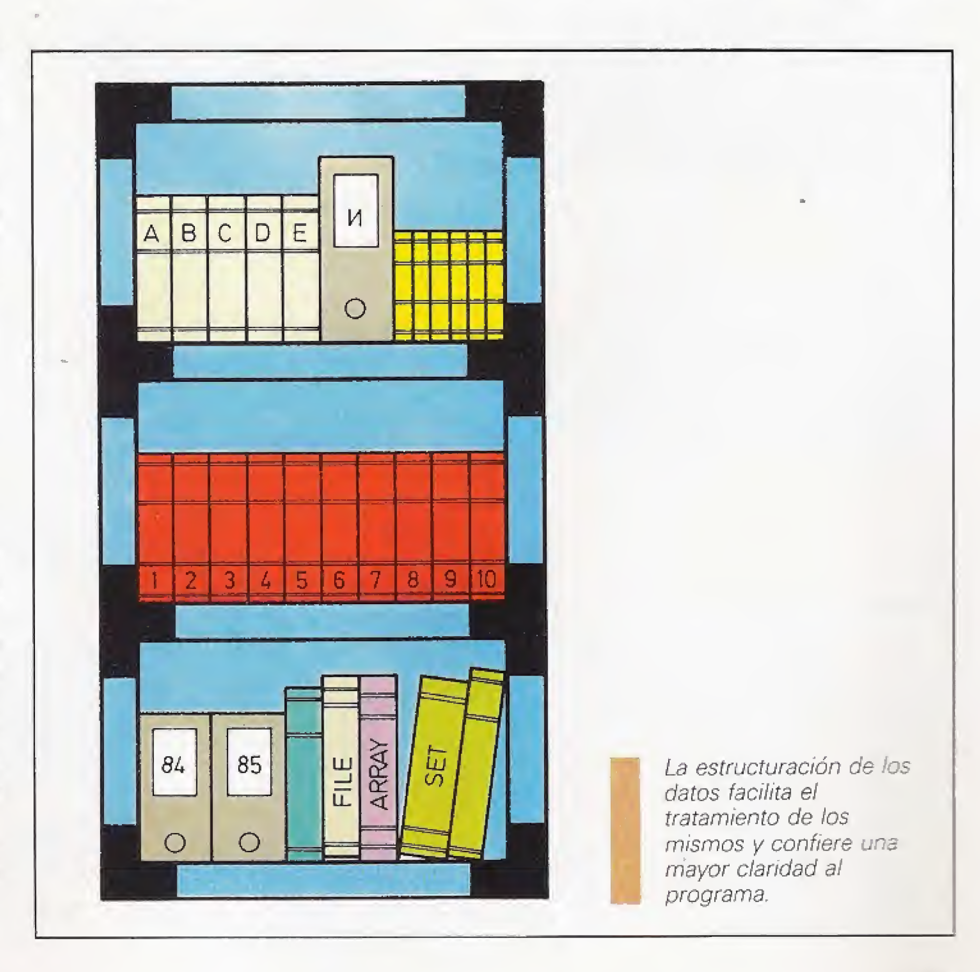
Por ejemplo, si se desea utilizar una variable para manipular los números de los días del mes, ésta se puede declarar como de tipo INTEGER. Pero, si por cualquier motivo se asigna a esta variable un valor entero inferior a «1» o superior a «31», obviamente se estará cometiendo un error de concepto que no será detectado por el programa; desde luego, a no ser que se codifique alguna sentencia que se encargue de comprobar tal situación. Esta sentencia no será necesaria si se define a la variable como de un tipo subrango de los números enteros, de la siguiente forma:

```
TYPE DIASDELMES: 1..31;
VAR DIA:DIASDELMES
```

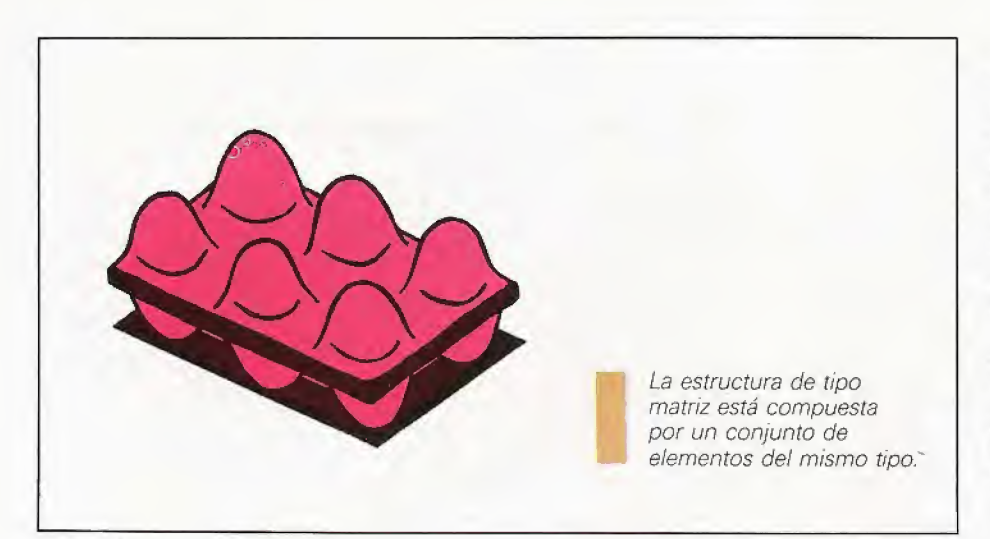
Ahora, la variable DIA sólo podrá tomar los valores de los números enteros comprendidos entre 1 y 31, ambos inclusive. Si se realizara una asignación fuera de ese rango —por ejemplo, DIAS:=40— se produciría un error. Como ya se mencionó, este tipo de da-



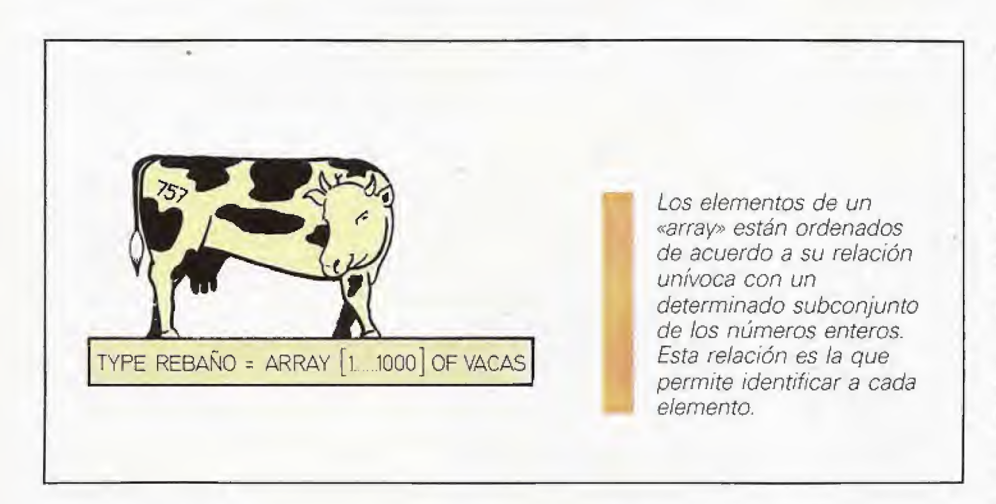

*Otra forma con la que cuenta el usuario para definir tipos de datos consiste en la selección de un subconjunto de valores de cualquier tipo escalar.*




*La estructuración de los datos facilita el tratamiento de los mismos y confiere una mayor claridad al programa.*



*La estructura de tipo matriz está compuesta por un conjunto de elementos del mismo tipo.*




*Los elementos de un «array» están ordenados de acuerdo a su relación unívoca con un determinado subconjunto de los números enteros. Esta relación es la que permite identificar a cada elemento.*

tos se puede definir a partir de cualquier otro tipo escalar, excepto del tipo REAL. Por lo tanto, serán válidas las definiciones de tipos que se realizan en el programa adjunto (CONCIENCIA).

## Datos estructurados

El fin primordial de la ciencia informática es el tratamiento de la información. Esta información toma cuerpo en los datos, que serán manipulados de acuerdo a los algoritmos apropiados para resolver cada problema específico.

En definitiva, los datos forman la componente estática del problema, mientras que la componente dinámica está constituida por los algoritmos que los manejan. En algunos casos, los datos serán fáciles de manipular y, por lo tanto, la componente dinámica del programa será también sencilla. Aunque la mayor parte de las veces no será así, sino que resultará obligado establecer algún tipo de relación entre los datos, que facilite su manejo y que simplifique la tarea de programación. Esto, ni más ni menos, hace patente la necesidad de estructurar los datos. Se define a una «estructura de datos», como una colección organizada de datos, en la que éstos están relacionados entre sí de alguna forma. Tal y como ocurría con la estructuración del programa, la estructuración de los datos presenta varias ventajas que justifican el estudio en profundidad de los datos a manejar en un programa. Entre otras contrapartidas, una estructura de datos bien pensada reduce enormemente la complejidad del programa que los va a manejar; por lo demás, aumenta la claridad de lectura y, en definitiva, mejora su seguimiento y facilita su posible modificación futura. Aún cabe citar otra razón más que aboga por el uso de datos estructurados: se consigue un mejor rendimiento al ejecutar el programa, debido a la menor cantidad de memoria necesaria para el almacenamiento de la información a procesar.

Dentro de sus virtudes como lenguaje de programación estructurado, el PASCAL permite la definición de cuatro tipos de datos estructurados: matrices (ARRAY), conjuntos (SET), registros (RECORD) y archivos (FILE).

## Matrices o arrays

Las matrices deben su existencia en los lenguajes de programación a la notación matemática que permite referirse a un conjunto de variables mediante un único vector de identificación, y seleccionar en él un valor mediante el empleo de subíndices.

Una matriz se define, pues, como un conjunto finito de componentes, todos del mismo tipo, que están ordenados en relación unívoca con un conjunto —también ordenado— de números enteros. Este conjunto de enteros ordenados recibe el nombre de subíndices. Cada uno de ellos permitirá el acceso a su componente correspondiente de la matriz.

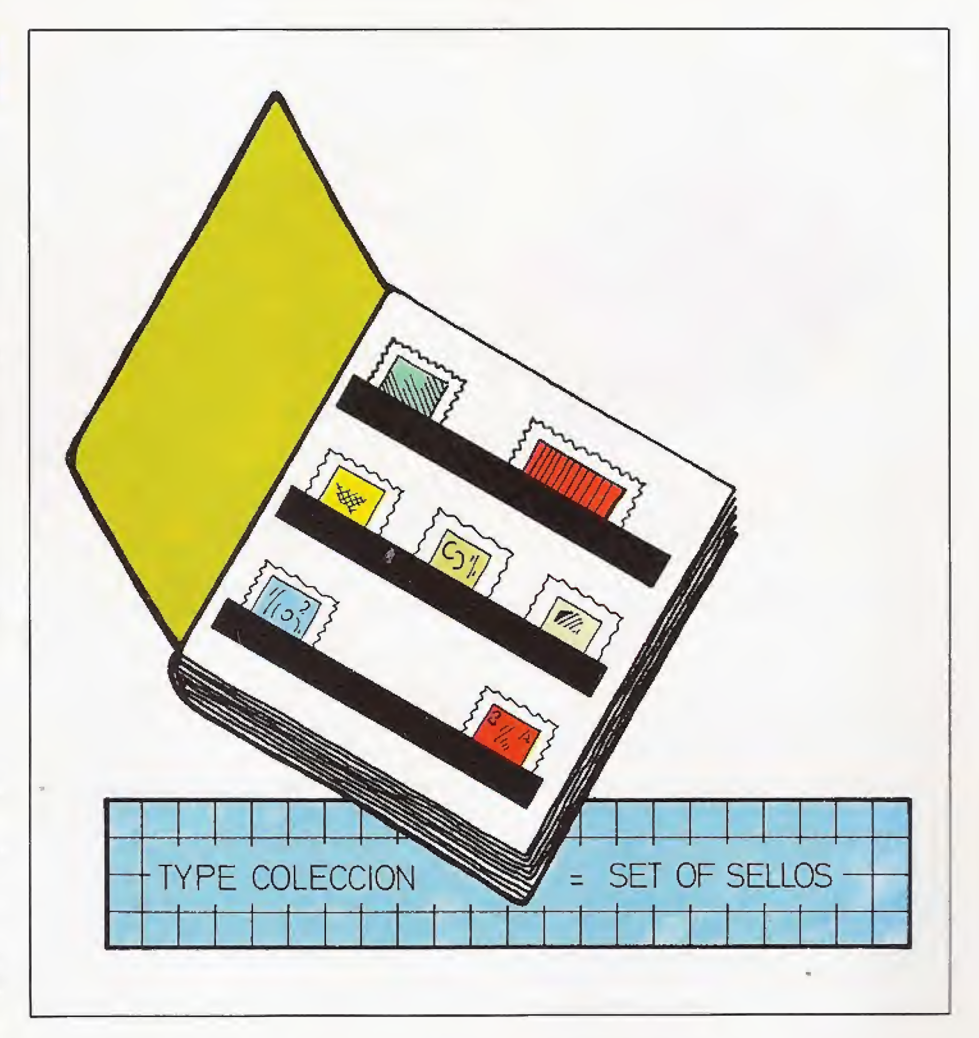
Esta misma idea es extensible a matrices n-dimensionales, en las que cada componente será referenciado por más de un subíndice. Así, para el caso particular de una matriz de dos dimensiones cuyo nombre sea «Z», se tiene:

$$Z=\begin{bmatrix} z11 & z12 & \dots & z1n \\ z21 & z22 & \dots & z2n \\ \dots & \dots & \dots & \dots \\ zm1 & zm2 & \dots & zmn \end{bmatrix}$$

Cada componente es identificado por una expresión del tipo Z[i, j], en donde el subíndice «i» indica la fila y el subíndice «j» indica la columna. En este caso, los subíndices i y j podrán adoptar los valores enteros comprendidos entre 1 y m, y entre 1 y n, respectivamente; con



*La estructura de tipo conjunto está integrada por una colección finita de objetos del mismo tipo.*



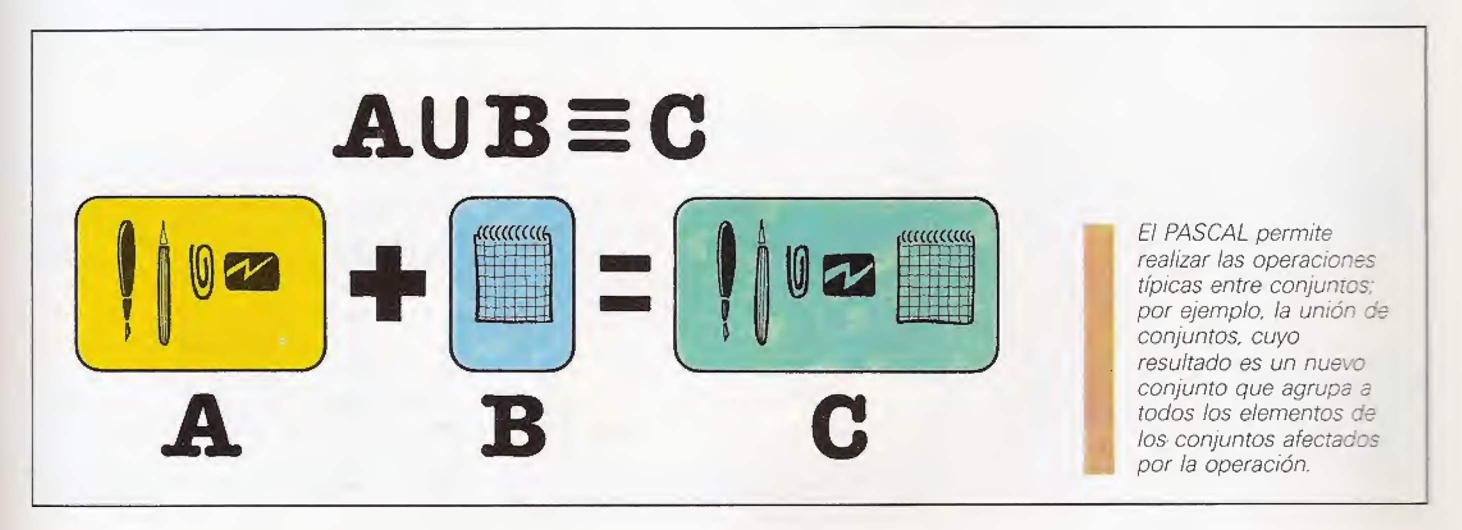

lo que cada componente queda unívocamente determinado. En PASCAL, el tipo de dato matriz se define con la siguiente notación:

```
TYPE <identificador>=ARRAY [<tipo indice>]OF <tipo
base>;
```

Por ejemplo:

```
TYPE tabla=ARRAY [1 ..100] OF REAL;
     caracter=ARRAY [CHAR] OF
     INTEGER;
```

En la definición cabe distinguir varias zonas. El <identificador>, coincidente con el nombre del nuevo tipo de dato definido. El <tipo índice>, que puede ser cualquier tipo ordinal (excepto el INTEGER) previamente definido (CHAR, BOOLEAN, enumerado o subrango); éste fijará el número de elementos que formarán la matriz. Hay que señalar que en el ejemplo de la declaración del tipo «tabla», aunque el tipo del índice pueda parecer que es INTEGER, por estar formado por los números enteros del 1 al 100, no es así, sino que se trata de un tipo subrango de los enteros. Finalmente, hay que hablar de la zona <tipo base>; ésta revela el tipo de todos y cada uno de los componentes (ya que todos son del mismo tipo), y puede ser cualquiera, incluso otro tipo estructurado. Es importante recordar que esta definición de tipo con TYPE, no declara variable alguna; en consecuencia, para




*El PASCAL permite realizar las operaciones típicas entre conjuntos; por ejemplo, la unión de conjuntos, cuyo resultado es un nuevo conjunto que agrupa a todos los elementos de los conjuntos afectados por la operación.*

poder utilizar este tipo de dato estructurado en un programa, habrá que declarar con VAR una variable matriz de un tipo ARRAY previamente definido. Por ejemplo, si se hace la declaración:

```
VAR x: tabla;
```

la nueva variable x será del tipo «tabla», que había sido definido en el ejemplo anterior. Así, la variable «x» quedará creada como una matriz de números reales con 100 componentes. No obstante, la declaración de variables matriz se puede realizar directamente de la siguiente forma:

```
VAR x: ARRAY [1..100] OF REAL;
```

No existe límite en la dimensión de las matrices; ello depende únicamente de la capacidad de memoria del ordenador utilizado.

La declaración de matrices multidimensionales se hace de la siguiente forma:

```
VAR parrafo:ARRAY [1 ..80] OF ARRAY
[1 ..20] OF CHAR;
```

O más abreviadamente:

```
VAR parrafo:ARRAY [1 ..80,1 ..20] OF CHAR
```

Como ya se indicó, a cada elemento de la matriz se accede por su nombre seguido de los subíndices encerrados entre corchetes; subíndices que incluso pueden ser expresiones. A los elementos de la primera declaración de párrafo se accederá, entonces, mediante :parrafo[i] [j]; mientras que a los de la segunda se hará con :parrafo[i,j]. Así, serán válidas las siguientes asignaciones, suponiendo declaradas las variables: «letra», «I» y «J»:

```
parrafo[8,12]:='A';
letra:=parrafo[50,10];
IF parrafo[I,J]=' ' THEN I:=I+1 ELSE
letra:='A';
```

Para aclarar mejor los conceptos relativos al uso de matrices de datos, veamos un sencillo ejemplo: un programa que realizará la multiplicación de dos matrices bidimensionales de números enteros (ver cuadro adjunto).

La operación de multiplicar matrices es específica de este tipo de datos, si bien se apoya en el producto y la suma de números enteros. De esta forma, el elemento [i,j] de la matriz resultante, se

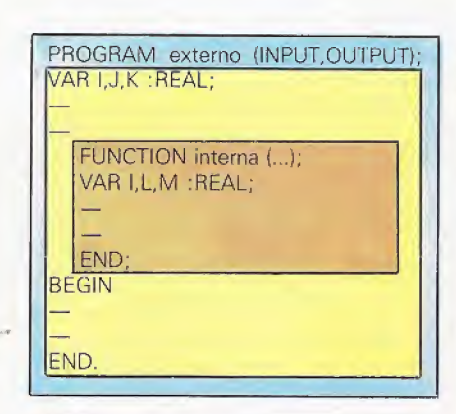
## Ambito de las variables

Para que una variable PASCAL pueda ser utilizada en el contexto de un programa, debe ser declarada con anterioridad su intervención. Esta declaración puede realizarse en distintas zonas de un programa: en la zona de declaraciones del propio programa, en la zona de declaraciones de una función o en la de un procedimiento. Ello hará que el ámbito de utilización de esa variable se vea restringido al bloque en el que ha sido declarada. Por ejemplo, si una variable se declara a nivel de programa, es decir en la zona que sigue al bloque PROGRAM, se podrá referenciar dentro de cualquier subprograma incluido en él, y también fuera de ellos. Este tipo de variables se denominan *globales*. Por el contrario, si una variable se declara dentro de una determinada función o procedimiento, ésta será de tipo *local:* sus valores sólo estarán disponibles dentro de dicho procedimiento o función y nunca en una zona del programa exterior al mismo. Así, por ejemplo, una variable puede ser global sólo en dos o más subprogramas, local en alguno de ellos o global a todos los subprogramas.

En el ejemplo de la figura, las variables J y K son globales a todo el programa cuyo nombre es «externo»; sus valores también estarán disponibles en el interior de la función denominada «interna». A su vez, las variables L y M son locales en la función «interna» y no serán reconocidas fuera de ella. Por el contrario, la variable I tendrá valores distintos fuera y dentro de la función «interna», ya que está declarada en ella como variable local y como variable global en el programa «externo».

La declaración de una variable local con el mismo identificador que una global no producirá error alguno, si bien los valores asignados a ella en cada zona del programa serán distintos. En efecto, aunque compartan el mismo nombre, representan a variables distintas. Esta característica suele ser fuente de numerosos errores por parte de programadores poco experimentados: es muy fácil intentar utilizar una variable local fuera de su ámbito, o bien querer disponer del valor de una variable global en un bloque en el que está definida otra variable local con el mismo identificador.

Para conocer el ámbito de una variable hay que examinar la zona de declaraciones del bloque en el que esté inmersa; si no está declarada en él, habrá que examinar el bloque que contenga al anterior, y así sucesivamente hasta llegar al bloque más general que coincidirá con el PROGRAM. Si tampoco está definida en él, resultará obvio que dicha variable generará un error de complicación al no estar declarada en ninguna zona del programa.




```
PROGRAM multiplicarmatrices (INPUT, OUTPUT);
CONST M1=6; M2=4; M3=2
VAR I: 1 . . M1; J: 1 . . M3; K: 1 . . M2;
    dato: INTEGER;
    A: ARRAY [1 . . M1,1 . . M2] OF INTEGER;
    B: ARRAY [1 . . M2,1 . . M3] OF INTEGER;
    C: ARRAY [1 . . M1,1 . . M3] OF INTEGER;
BEGIN
      FOR I:=1 TO M1 DO
      BEGIN
            FOR K:=1 TO M2 DO
            BEGIN
                  READLN(dato);
                  A[I,K]:=dato
            END
            WRITELN
      END;
      FOR K:=1 TO M2 DO
      BEGIN
            FOR J:=1 TO M3 DO
            BEGIN
                  READLN(dato);
                  B[K,J]:=dato
            END;
            WRITELN
      END;
      FOR I:=1 TO M1 DO
      BEGIN
            FOR J:=1 TO M3 DO
            BEGIN
                  DATO:=0;
                  FOR K:=1 TO M2 DO
                        dato:=dato+A [I,K] * B[K,J];
                        C[I,J]:=dato;
                        WRITE(dato)
            END;
            WRITELN
      END
END.
```

obtiene sumando los productos ordenados de los términos de la fila i-ésima de la primera matriz, por los términos de la columna j-ésima de la segunda matriz. Para ello, se dimensionan como constantes el rango de enteros que determinará el tamaño de las matrices, almacenándose éstas en las matrices A y B. A su vez, el resultado se almacenará en la matriz C.

Para multiplicar matrices de otras dimensiones, sólo habrá que modificar los valores constantes declarados con CONST.

## El tipo SET

El lenguaje PASCAL permite manejar el concepto matemático de «conjuntos», característica ésta que no comparten la mayoría de los restantes lenguajes de programación. En todo caso, cabe señalar que su uso se ve reducido a casos muy particulares. Un conjunto se puede definir como: «una colección de elementos distintos entre sí, pero del mismo tipo».

Los conjuntos se representan en PASCAL escribiendo sus elementos entre corchetes:

```
[1,3,7,9,21,34,11]
[rojo,verde,azul]
```

Las variables de conjuntos, es decir, aquellas que se declaren como de algún tipo SET, tal y como se verá más adelante, pueden contener en un momento dado ninguno, uno, varios o todos los

### Funciones estándar de los tipos ordinales

Todos los tipos de datos analizados hasta ahora pertenecen a la categoría general de «tipos ordinales». Ello se debe a que, para cualquier valor dado, existe siempre un sucesor y un predecesor del mismo, salvo que se trate del primer o último elemento de alguno de estos tipos.

El lenguaje PASCAL dispone de dos funciones estándar que permiten obtener los valores anterior o posterior a un dato:

PRED(X) y SUCC(X); en donde X es un valor de cualquier tipo ordinal.

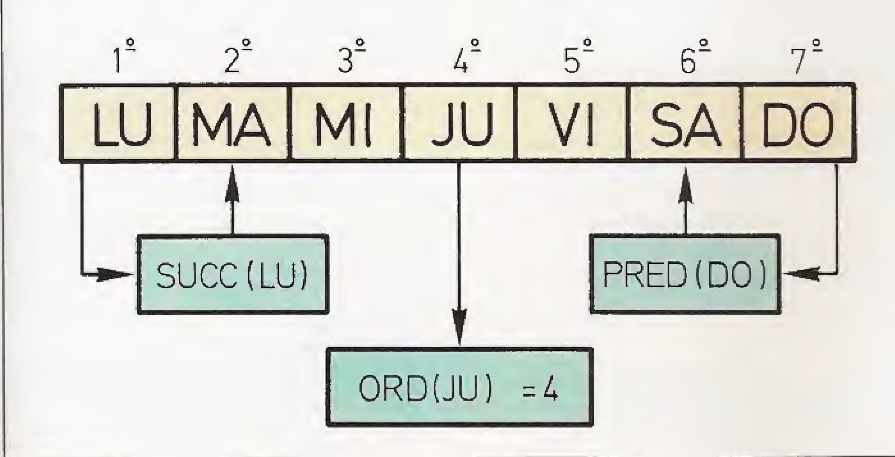
Supóngase, por ejemplo, que se ha definido el tipo enumerado siguiente:

```
TYPE
  DIASSEMANA=(LUN,MAR,MIE,
    JUE,VIE,SAB,DOM);
```

La función SUCC(LUN) dará como resultado MAR, mientras que la función PRED(SAB) entregará el valor VIE. Otra función definida para los tipos ordinales es ORD(X). Su objetivo es proporcionar el número entero que indica el orden que ocupa el valor X dentro del conjunto de posibles valores del tipo ordinal al que pertenece.

Otra función estándar, sólo aplicable a datos del tipo CHAR, es CHR(I). Esta devuelve el carácter cuyo número ordinal coincide con el valor indicado por el parámetro de entrada I. Así, la expresión INICIAL:=CHR(ORD('A')+I–1), asignará a la variable INICIAL la letra I-esima del abecedario.

Todas estas funciones permiten proyectar los valores de cualquier tipo ordinal sobre un conjunto de los números naturales y actúan, en consecuencia, como virtuales funciones de transferencia.




B

4 2 7 1 9 9 5 8 3

VAR B: SET OF INTEGER;

[1] IN B = TRUE

*La función IN comprueba la relación de inclusión entre elementos y conjuntos; esto es, verifica la pertenencia de un elemento a un conjunto.*

```
PROGRAM CONCIENCIA (INPUT);
TYPE
        DIASSEMANA=(LUN,MAR,MIE,JUE,VIE,SAB,DOM);
        LABORABLES=LUN..VIE;
        CANTIDAD=1..5;
        LETRA='A'..'Z';
VAR     HOY:DIASSEMANA;
        COPAS:CANTIDAD;
        CURRAR:LABORABLES;
        NUM:INTEGER;
        INICIAL:LETRA;
BEGIN
        READLN (INICIAL);
        CASE INICIAL OF
        'L':HOY:=LUN;
        'M':HOY:=MAR;
        'X':HOY:=MIE;
        'J':HOY:=JUE;
        'V':HOY:=VIE;
        'S':HOY:=SAB;
        'D':HOY:=DOM
        END;
        CURRAR:=LUN;
        REPEAT BEGIN
          IF HOY=CURRAR THEN COPAS:=1
          ELSE COPAS:=5;
          CURRAR:=SUCC(CURRAR)
        END
        UNTIL CURRAR:=VIE OR HOY=CURRAR;
        WRITELN('NO BEBER MAS DE ',COPAS,' COPAS')
END.
```

| TABLA DE COMANDOS-PASCAL | | |
|---|---|---|
| **Comando** | **Cometido** | **Observaciones** |
| FUNCTION <id.>{(<lista>)}:<tipo>; | Declaración como función de un conjunto de sentencias | Zona de declaraciones y definiciones |
| <id.1>,...:<tipo.1> ,<id.2>,...:<tipo.1>,... | Declaración de los parámetros formales de una función | Cabecera de una FUNCTION |
| TYPE <id.> = <tipo>; {...} | Declaración de tipos de datos no estándar | Zona de declaraciones y definiciones previa a VAR. |
| TYPE <id.> = (<id.1>,...,<id.n>); | Declaración de tipos por enumeración de datos | Zona de declaraciones y definiciones previa a VAR |
| TYPE <id.> = <cte.>..<cte.>; | Declaración de tipos subrango de datos | Zona de declaraciones y definiciones previa a VAR |
| PRED (X); | Función que calcula el predecesor de un dato ordinal | Cuerpo del programa |
| SUCC (X); | Función que calcula el sucesor de un dato de tipo ordinal | Cuerpo del programa |
| ORD (X); | Función que calcula el número de orden de un dato de tipo ordinal | Cuerpo del programa |
| CHR (I); | Función que calcula el carácter correspondiente al número ordinal I | Cuerpo del programa |
| ARRAY [<tipo índice>] OF <tipo base>; | Definición del tipo matriz | Zona de definición de tipos con TYPE |
| SET OF <tipo2>; | Definición del tipo conjunto | Zona de definición de tipos con TYPE |
| <conj.>*<conj.> | Operación de intersección de conjuntos | Cuerpo del programa |
| <conj.>+<conj.> | Operación de unión de conjuntos | Cuerpo del programa |
| <conj.>-<conj.> | Operación de diferencia de conjuntos | Cuerpo del programa |
| <elem.> IN <conj.> | Relación de pertenencia o inclusión en un conjunto | Cuerpo del programa |
| <conj.> <=<conj.> | Relación de inclusión entre conjuntos | Cuerpo del programa |

NOTA: <id.>, <id.1>, ..., <id.n>: identificadores válidos PASCAL. <lista>: lista de parámetros formales. <tipo>: cualquier tipo de datos. <tipo1>: tipo de dato no estructurado. <cte.>: constante de cualquier tipo excepto REAL. (X): dato de tipo ordinal. (I): número natural. <tipo índice>: cualquier tipo ordinal, excepto INTEGER. <tipo base>: cualquier tipo de dato incluso otro estructurado. <tipo2>: tipo de dato escalar o subrango, excepto REAL. <conj.>: conjunto de un SET. <elem.>: elemento de un SET.

elementos que forman parte de dicho conjunto. Por lo tanto, habrá tantos posibles valores como resulte de elevar la base 2 a la potencia coincidente con el número de objetos que componen el conjunto.

El aspecto general de una definición del tipo conjunto viene dada por:

```
TYPE <identificador>= SET OF <tipo base>;
```

Por ejemplo:

```
TYPE numero= SET OF 0..9;
    palabra=SET OF CHAR;
```

El <tipo base> puede ser de cualquier tipo escalar o subrango, excepto de tipo REAL. Como siempre, la definición de tipo no declara la variable, por lo que ésta debe declararse aparte, o bien directamente:

```
TYPE colores=(rojo,verde,azul,amarillo,blanco,negro);
VAR cuadro: SET OF colores;
    cuadro2: SET OF (rojo,verde,azul);
```

Cada elemento de tipo SET se referencia por medio de un subrango de valores: [1..5], o bien por un subconjunto de esos valores separados por comas: [1,2,3,4,5]. Para no complicar en exceso el cálculo de valores por parte del compilador, ni ocupar excesiva memoria, y debido al uso esporádico de este tipo de datos, se suele limitar el número de elementos del conjunto. Un valor típico se concreta en un máximo de 64 elementos.

Para este tipo de datos existen una serie de operadores específicos. Estos realizan las operaciones básicas entre conjuntos definidas por la teoría matemática moderna. Así, para la unión de conjuntos se emplea el símbolo «+», para la intersección el símbolo «*», y para la diferencia el símbolo «–». Las siguientes operaciones darán como resultado el que se indica:

```
[azul,rojo]+[rojo,verde]=[azul,rojo,verde]
[azul,rojo]*[rojo,verde]=[rojo]
[azul,rojo]-[rojo,verde]=[azul]
```

Además de los operadores definidos, se pueden aplicar a elementos de tipo SET los operadores de relación ya conocidos: =, <>, <=, >=. Estos últimos darán como resultado un valor lógico TRUE o FALSE.

Los operadores: '<='y'>=' no tienen el sentido de comprobar si un conjunto es mayor o menor que otro, ya que ello es absurdo, sino que tienen la función de comprobar si un conjunto está o no incluido en otro. Así, por ejemplo: la expresión lógica [verde]<=[azul,verde], dará como resultado TRUE al ser evaluada. Para comprobar la pertenencia o inclusión de conjuntos existe un operador específico. Este es el operador binario «IN», que obtiene un resultado de tipo lógico (TRUE o FALSE). Así, la expresión: [verde] IN [rojo,azul] adoptará el valor FALSE, mientras que: [verde] IN [rojo,verde] será TRUE.

Los tipos RECORD y FILE, por su importancia, se estudiarán con más detalle y detenimiento en el próximo capítulo de la obra.

# PASCAL

## Otras estructuras de datos



**A**nalizadas ya las ventajas de la estructuración de los datos y estudiados los dos primeros tipos de datos estructurados —matrices y conjuntos (ARRAY y SET)—, sólo nos quedan por examinar los dos últimos tipos: registros y archivos o ficheros. El tipo *registro* (RECORD) es el más flexible de todos, ya que permite reunir datos de cualquier otro tipo, incluso nuevos registros, dentro de una misma estructura. La denominación de «registro» está presente con harta frecuencia en la vida cotidiana, dentro de frases como: registro de empleados, oficina de registros, registro de la propiedad..., lo que da una idea de su gran utilidad.

En PASCAL, el concepto de registro se refiere a un conjunto de informaciones relativas a un único objeto, que pueden ser utilizadas como un «todo agrupado», o referenciando a sus partes individualmente. Estas partes, denominadas *campos,* mantienen una cierta independencia unas respecto a otras, lo que permite su diferenciación clara en tipo y contenido. A título de ejemplo, se puede pensar en la información reflejada en un carnet de identidad que constituye, por sí mismo, un «registro» con la identidad de una persona:

— Número de orden
— Nombre
— Apellidos
— Lugar de nacimiento
— Fecha de nacimiento
— Nombres de los padres
— Estado civil
— Profesión
— Domicilio
— Grupo sanguíneo

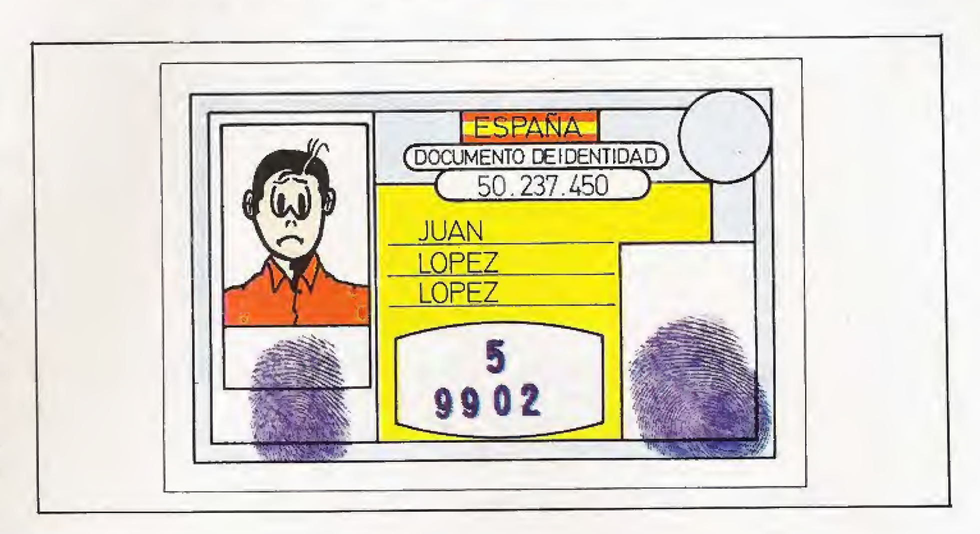
Este registro, que da lugar al carnet de identidad de una persona, está integrado por 10 campos; cada uno de ellos puede contener informaciones cuyo tipo difiere de la incluida en los restantes campos.

Obviamente, el tipo de información que se almacene en el campo del nombre del sujeto no tendrá ninguna semejanza con el contenido del campo de la fecha de nacimiento. Este último estará compuesto, a su vez, por otros tres datos que indicarán el día, el mes y el año, mientras que aquél contendrá simplemente una cadena de caracteres. Esta es una de las principales diferencias del tipo registro (RECORD) respecto al tipo ARRAY. La otra diferencia importante reside en que mientras en una matriz o array cada uno de los elementos se referencian por un subíndice, en los registros se hace referencia a sus elementos por el nombre de los campos que lo integran.

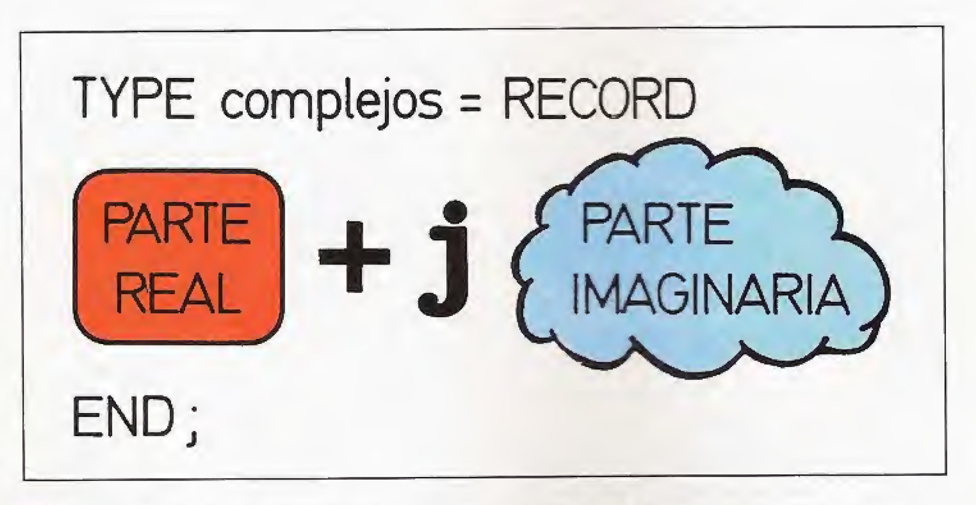
Un poco más formalmente, se puede definir un registro en PASCAL como una colección de un número fijo de componentes denominados «campos», estando cada campo definido por un identificador y su tipo de dato correspondiente (tipo que puede ser distinto para cada campo). La definición de un dato de tipo registro se realiza de la siguiente forma:

```
TYPE <identificador>=RECORD
     <lista de campos>
     END;
```

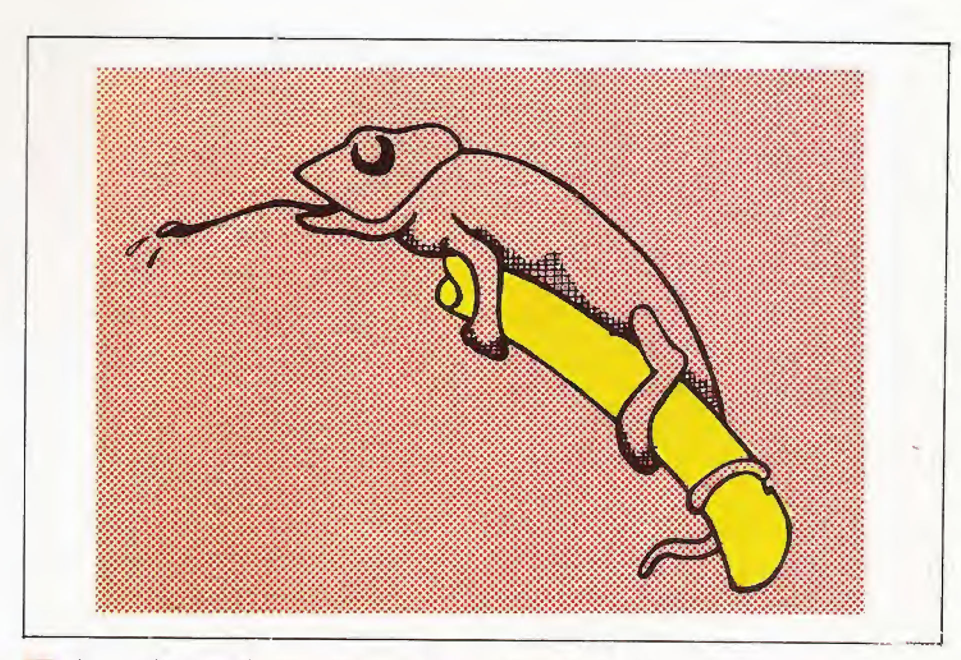
Donde la lista de campos estará compuesta por un número finito de identificadores de campo, seguidos por el tipo




*El tipo RECORD (registro) permite crear una estructura con varias «campos» o componentes que pueden ser de muy diversos tipos.*




*Las estructuras de tipo RECORD resultan de gran ayuda en múltiples situaciones; por ejemplo, cuando hay que trabajar con números complejos.*



*Los registros variantes acomodan el número y el tipo de sus campos a las circunstancias actuales, al igual que un camaleón acomoda su color al del terreno que lo circunda.*

de cada campo —éste puede ser cualquier tipo estándar o definido con anterioridad—, y separados por el signo de punto y coma. Esta es, ni más ni menos, la estructura genérica:

```
<identificador campo 1>:<tipo>;
<identificador campo 2>:<tipo>;
...
<identificador campo n>:<tipo>;
```

Veamos un ejemplo:

```
TYPE fecha=RECORD
     día: (1..31);
     mes: (ENE, FEB, MAR, ABR, MAY,
     JUN, JUL, AGO, SEP, OCT, NOV,
     DIC);
     año: (1900..2000)
     END;
TYPE complejos=RECORD
     partereal:REAL;
     parteimag:REAL
     END;
```

Las palabras «fecha» y «complejos» son los identificadores de los registros definidos, mientras que «día», «mes», «partereal», etc., son los identificadores de los campos.

## Seleccionando los campos

Como ya se ha mencionado, los elementos de un registro se referencian mediante el identificador de los campos, a través del llamado selector de campo. Este no consiste más que en el identificador de la variable RECORD seguido por un punto y por el identificador del campo.

Un registro puede utilizarse tanto en definiciones de tipo como en declaraciones de variables; así, si se declara la variable «fechanacim» como del tipo «fecha» (definido en el ejemplo anterior) mediante la correspondiente sentencia VAR, se podrá hacer referencia a sus campos respectivos mediante los siguientes selectores de campos:

```
fechanacim.día
fechanacim.mes
fechanacim.año
```

De esta forma, cualquier campo de un registro puede ser utilizado como si se tratara (de hecho lo es) de una variable normal. Pero no sólo se pueden utilizar los campos por separado, sino que además se puede hacer referencia al registro en bloque, ya sea dentro de sentencias de asignación, como argumento de funciones, etc. Veamos el siguiente ejemplo:

```
TYPE complejos=RECORD
         partereal:REAL;
         parteimag:REAL
     END;
VAR reactancia: complejos;
    inductancia:complejos;
BEGIN
rectancia.partereal:=0;
reactancia.parteimag:=0;
inductancia:=reactancia;
...
END.
```

La última sentencia de asignación copiará en su integridad el registro «reactancia» en el registro «inductancia». El efecto sería el mismo si se tratara de registros con mayor número de campos y con los tipos de los campos más complicados (estructurados).

Una sentencia que ayuda a manejar los registros es WITH, cuyo formato es el siguiente:

```
WITH <variable registro> DO <sentencia>;
```

Con ella se puede omitir el identificador del registro, indicando únicamente el identificador del campo que se desea seleccionar. De esta forma se abreviará sustancialmente la escritura de un programa, como puede observarse en el siguiente ejemplo:

```
WITH reactancia DO
     BEGIN
          partereal:=4;
          parteimag:= -3.9;
          modulo:=SQRT(SQR(partereal)+SQR(partei-
     mag));
          WRITELN(modulo)
     END;
```

La sentencia WITH sirve de gran ayuda en el caso de tener registros de estructura compleja, en los que alguno de sus campos sean a su vez nuevos registros (registros jerarquizados).

## Registros variantes

La estructura de las variables ordinarias de tipo registro es bastante rígida;

de tal forma que todos los registros de un mismo tipo tendrán el mismo número de campos, con los mismos nombres y con los mismos tipos.

En muchos casos puede ser conveniente hacer que esta estructura resulte un poco más moldeable, con objeto de adecuarla a las posibles variaciones de los datos en cuanto al número y tipo de los campos. Esto se consigue mediante los llamados registros variantes, cuyo método de definición es el siguiente:

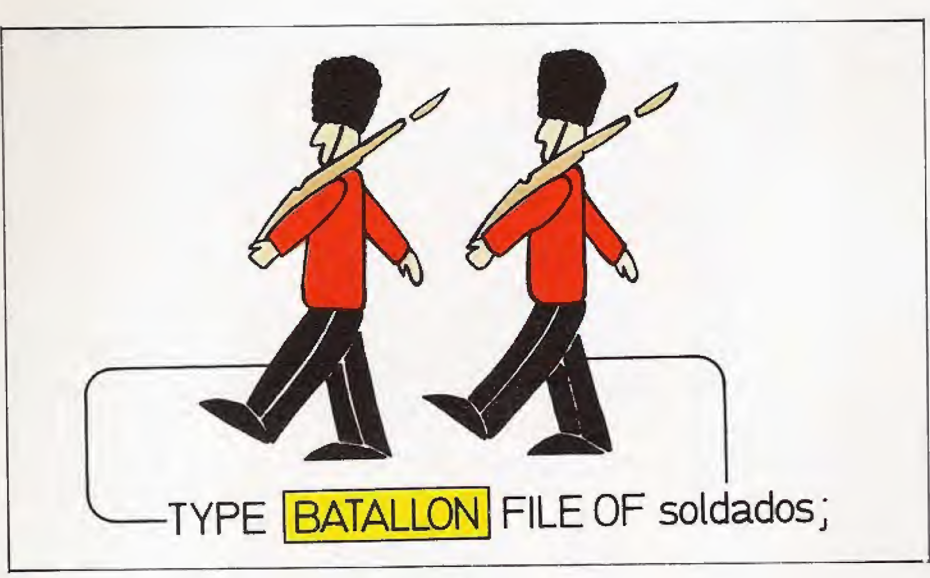
```
TYPE <identificador>=RECORD
     <campo1>:<tipo>;
     <campo2>:<tipo>;
     ...
     CASE <campo distintivo>:
          <tipo> OF
          <valor1>:(<id.1>:<tipo>;
          <id.2<:<tipo>;...)
          <valor2>:(<id.1>:<tipo>;
          <id.2>:<tipo>;...)
          ...
     END;
```

Por ejemplo:

```
TYPE datos=RECORD
     nombre, apel1, apel2:PACKED
     ARRAY [1..20] OF CHAR;
     nacim:fecha;
     nacionalidad:(ESPAÑOLA,
     EXTRANJERA);
     CASE civil:(SOLTERO,
          CASADO) OF
          SOLTERO:(novia:BOLEAN);
          CASADO:(boda:fecha;mujer
          ARRAY[1..20] OF CHAR)

     END;
```

El denominado «campo distintivo» actúa como seleccionador de la sentencia CASE, la cual determina el número y tipo de campos requeridos en cada circunstancia según su valor actual; ello proporciona una mayor flexibilidad a los datos de tipo RECORD.




*El tipo FILE (fichero) consta de una sucesión de componentes del mismo tipo.*




*La variable «buffer» de un fichero se utiliza a modo de «ventana» a través de la cual se puede leer o escribir un elemento del fichero.*

## Los archivos de datos

En muchas ocasiones resulta útil disponer de medios para crear un conjunto de datos que pueda ser utilizado en otro momento por el mismo ordenador, o incluso por otro ordenador diferente. Este conjunto de datos se suele almacenar en los llamados «archivos» o «ficheros»: estructuras formadas por una secuencia de componentes, todos del mismo tipo, de forma que sólo se puede acceder a un único componente del mismo en un momento dado. El acceso a un elemento se efectúa recorriendo previamente y en secuencia los restantes elementos que estén por delante del afectado. Lo indicado equivale a trabajar con una estructura secuencial en la que importa el lugar físico en el que se almacenan los componentes y que, generalmente, será una memoria de masa externa (cinta magnética o disco).

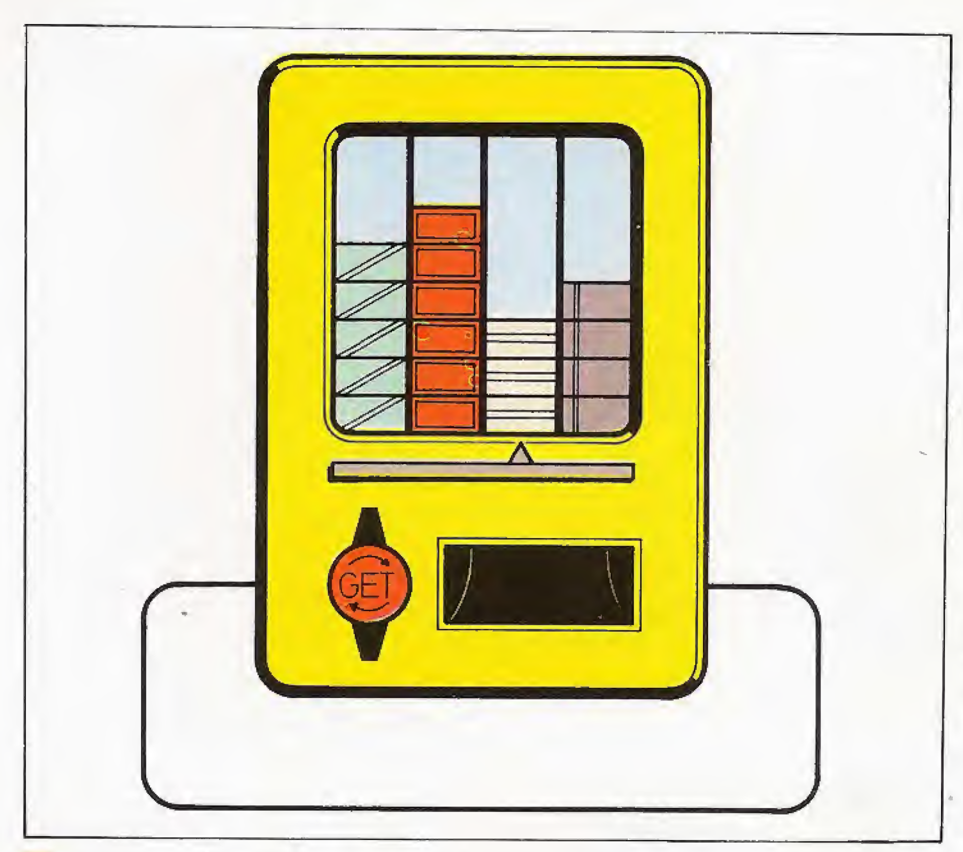
Al hablar de los ficheros estándar de entrada y salida, se comentó la forma general de definir las variables de tipo fichero. Esta era, como se recordará:

```
TYPE <identificador>=FILE OF <tipo de los elementos>;
```

Por ejemplo:

```
VAR numeros=FILE OF complejos;
```

En donde el tipo de los elementos puede ser cualquiera, incluso un tipo




*El procedimiento GET extrae, uno a uno, los sucesivos elementos de una variable de tipo fichero.*

estructurado que haya sido previamente definido. Las variables de tipo fichero se pueden utilizar como parámetros de procedimientos, pero no en sentencias de asignación ni de otro tipo. Para facilitar el manejo de los ficheros existen una serie de procedimientos estándar definidos previamente por el compilador de PASCAL.

Uno de estos procedimientos es el que tiene por sentencia de llamada: RESET (<identificador de fichero>). Su misión es preparar el fichero para que pueda ser leído. La lectura podrá realizarse entonces mediante la ya conocida sentencia READ(<identificador de fichero>,x). El valor contenido en la componente actual del fichero pasa a almacenarse en la variable auxiliar «x»; ésta debe ser, por tanto, del mismo tipo que las componentes del fichero. El valor de dicha componente estará ahora disponible en la variable «x» para utilizarlo en la forma que se desee. Para acceder a la siguiente componente se empleará una nueva sentencia READ, y así sucesivamente hasta que se llegue a la última componente del fichero, en cuyo caso, una nueva lectura con READ producirá un error debido a que no existen más componentes. La condición de fin de fichero se puede examinar previamente, para evitar estos errores, mediante otro procedimiento estándar, cuya sentencia de llamada es EOF (<identificador de fichero>). Este procedimiento devuelve el valor booleado TRUE si se ha llegado al final del fichero; de lo contrario, el valor devuelto será FALSE.

La lectura de todas las componentes del fichero «numeros», previamente declarado, se puede programar de la siguiente forma:

```
WHILE NOT EOF(numeros) DO
    BEGIN
        READ(numeros,x);
    ...
    END;
```

Para inicializar la escritura de datos en un fichero se dispone del procedimiento REWRITE (<identificador de fichero>); cuya misión es sustituir el fichero indicado por un nuevo fichero vacío y posicionarse en la primera componente. La escritura se realizará entonces mediante una sentencia WRITE (<identificador de fichero>,x); habiendo asignado previamente a la variable auxiliar «x» el valor que se quiere introducir en la componente actual del fichero. Así, sucesivamente, se irán almacenando todas las componentes del mismo.

Para los ficheros de texto existe un procedimiento que indica cuando se ha alcanzado el final de una línea; éste se activa mediante la llamada EOLN (<identificador de fichero>) y devuelve un valor de tipo booleano de forma similar al procedimiento EOF. La declaración de cada variable de tipo fichero crea automáticamente una variable auxiliar denominada «buffer», con cabida para un único valor del mismo tipo que las componentes del fichero; ésta variable se utiliza como «intermediaria» en la lectura y escritura de valores.

La variable auxiliar «buffer» está a disposición del programador en cualquier momento, apelando al identificador: <nombre del fichero>↑.

Así, en el ejemplo del fichero «numeros», creado anteriormente, la variable «buffer» se referenciará por medio de: numeros↑. Esta variable también es utilizada por otros dos procedimientos estándar: GET (<identificador de fichero>) y PUT (<identificador de fichero>) que constituyen otra forma de actualizar las componentes del fichero. Así, GET obtiene el valor de la siguiente componente y lo almacena en la variable buffer del fichero, mientras que PUT traslada el valor del buffer a la componente actual del fichero; lógicamente, ello sólo tendrá sentido cuando EOF sea TRUE, debido a la estructura secuencial del tipo fichero.

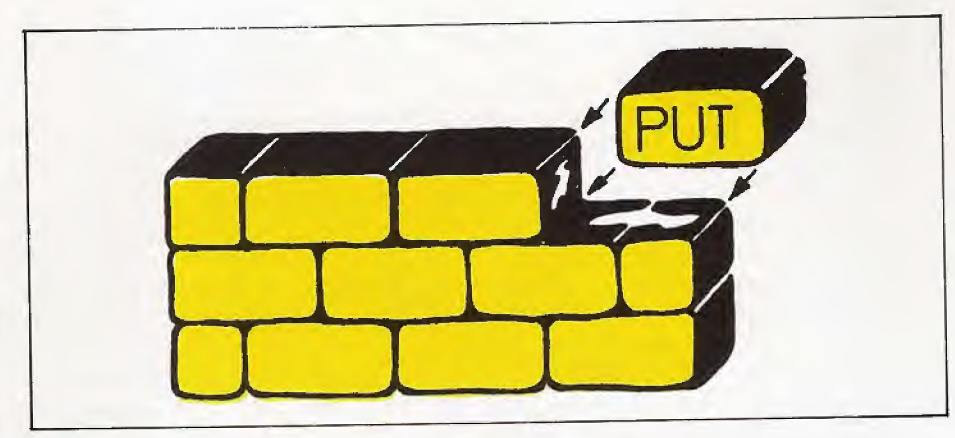
## Datos «empaquetados»

La importancia e indudable utilidad del tipo registro (RECORD) estudiado, da pie a describir una serie de conceptos relacionados más o menos directamente con el mismo. Por ejemplo, la forma

en que se aprovecha la memoria interna que brinda el ordenador no es un tema relacionado directamente con lenguajes de alto nivel, si bien, cobra cierta importancia cuando éstos son capaces de influir en ella.

El elemento básico de información de cara a su tratamiento por parte del ordenador y a su almacenamiento en memoria, es la palabra binaria. Cada una de estas palabras representará a un dato de tipo simple (número, carácter, valor lógico...) y ocupará una de las posiciones o celdas de la memoria interna. Como quiera que lo habitual es trabajar con palabras binarias de 8 bits, cada una de dichas celdas estará integrada por ocho casillas o elementos de memoria. Esta forma de almacenamiento se denomina desempaquetada (UNPACKED), y constituye el método más natural de guardar la información.

Cada dato ocupará, en consecuencia, una posición de memoria. Sin embargo, si en una posición de memoria (integrada por ocho celdillas de bit) es posible almacenar, por ejemplo, datos numéricos de hasta cinco cifras, no cabe duda que al almacenar un dato con tan sólo dos cifras se estará desaprovechando el espacio de memoria. Si esa información




*PUT es el procedimiento PASCAL que realiza la función contraria a GET; esto es, introduce un nuevo elemento en la última posición del fichero.*

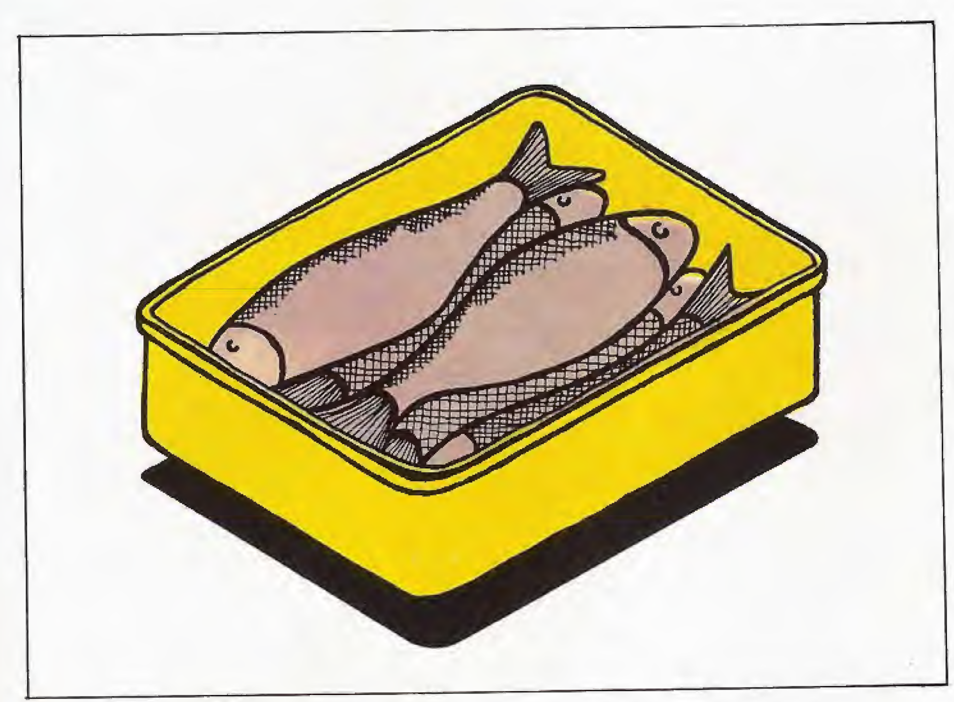
se pudiese «empaquetar» de alguna forma, aumentaría el rendimiento de la máquina.

El PASCAL permite dicho empaquetado de la información mediante el atributo PACKED, aplicable a las estructuras RECORD y ARRAY:

```
TYPE fecha=PACKED RECORD
      dia:1..31;
      mes:1..12;
      año:0..2000
      END;

      tira=PACKED ARRAY[1..15]
      OF CHAR;
VAR hoy:fecha;
      nombre:tira;
```

Así, por ejemplo, en la variable «hoy» el número correspondiente al día se empaquetará junto con el número del mes en una sola palabra. A su vez, en otra palabra se guardará el número del año. De esta forma tan sólo son necesarias dos palabras, mientras que si el almacenamiento se realizara en modo desempaquetado se necesitarían tres palabras: una para cada uno de los números correspondientes al día, al mes y al año. Para registros con campos más complicados, el compilador de PASCAL se encarga de realizar el empaquetamiento de los mismos de la forma más eficaz.



*El atributo PACKED permite un mejor aprovechamiento de la memoria al «empaquetar» los datos de la forma más conveniente a cada caso.*

## Cadenas de caracteres (strings)

Un tipo de dato utilizado con harta frecuencia en los programas es el «string» o «cadena de caracteres». Este se define como una matriz empaquetada de caracteres de la forma que se indica en el ejemplo anterior. Su uso permitirá manejar una secuencia de caracteres que representen, por ejemplo, nombres de personas u objetos, ya que como se re-

```
VAR NOMBRE = ARRAY [1.....20] OF CHAR;
NOMBRE : = 'FRANCISCO'
```

*Mediante la definición de un tipo ARRAY de caracteres es posible manejar con comodidad y eficacia las denominadas cadenas de caracteres o «strings».*

cordará las variables de tipo CHAR sólo podían almacenar un único carácter.

Una de las operaciones más frecuentes con este tipo de datos es la de clasificarlos por orden alfabético; para ello se pueden utilizar los operadores lógicos de relación: <,>,=,<>, <=,>=... Una ventaja de las cadenas de caracteres es que permiten asignar un valor a todos sus elementos simultáneamente, obviando la necesidad de realizar esta asignación carácter por carácter. Una limitación al respecto es que el número de caracteres debe coincidir con el tamaño declarado para la cadena, no será posible realizar la asignación desde el exterior mediante una sentencia de tipo READ. En tal caso, será necesario hacerlo carácter a carácter, por ejemplo, mediante bucles FOR. No obstante, sí será posible proyectar una cadena completa al exterior mediante la correspondiente sentencia WRITE, siendo el tamaño del campo del formato de salida igual al tamaño de la cadena de caracteres a extraer.

## Punteros

El lenguaje PASCAL posee un tipo de variables denominadas «punteros» o «apuntadores» (POINTERS), cuya misión es almacenar la dirección de memoria de un determinado dato. Con su colaboración es posible enlazar los datos de forma que la cantidad de memoria ocupada por un determinado conjunto de datos sea únicamente la imprescindible, ni más ni menos. Esta característica se conoce como estructura dinámica de datos. Existen procedimientos que permiten enlazar unos datos con otros, borrarlos o adicionarlos; y todo ello sin necesidad de volver a reorganizar toda la estructura ni definir previamente el tamaño necesario, como ocurriría con las variables de tipo ARRAY, en el caso que se pretendiera utilizarlas para producir un número de elementos no determinado de antemano.

Las variables de tipo puntero se definen de la siguiente forma:

```
TYPE <identificador>: ^ <tipo>;
```

siendo el signo «^» el que diferencia a este tipo de dato, y <tipo> el parámetro que señala el tipo de datos a los que apuntarán estas variables (puede ser cualquier tipo, incluso estructurado).

Para inicializar la variable puntero y reservar así espacio de memoria para la primera variable, se dispone del procedimiento: NEW (<variable apuntador>);

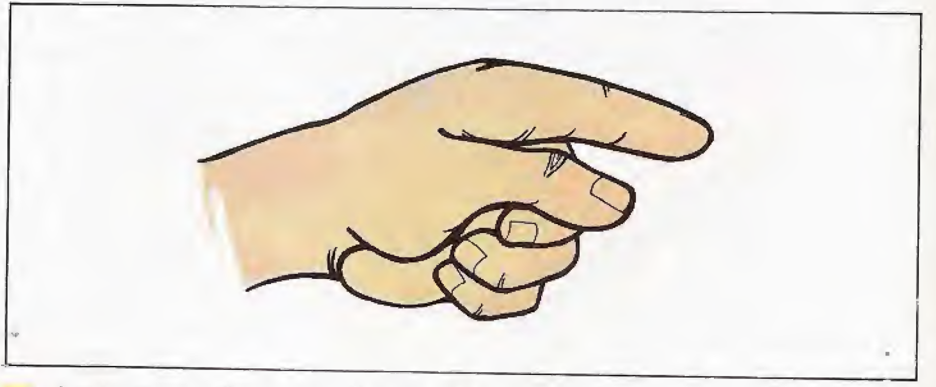
La operación contraria corre a cargo del procedimiento: DISPOSE (<variable apuntador>);

Para acceder a la posición reservada se hace uso de la notación siguiente: <variable apuntador>; ^ para pasar a la posición siguiente se ejecutará un nuevo NEW. Si se desea que la variable puntero no señale a ningún elemento, habrá que asignar a ésta el valor NIL.

Mediante la definición de los registros adecuados con algún campo de tipo puntero, será posible crear listas encadenadas de datos con carácter dinámico que, evidentemente, potenciarán el rendimiento del ordenador.

## PASCAL: el lenguaje de los 80. Repasando conceptos.

Como habrá podido comprobarse, el PASCAL es un lenguaje de programación muy atractivo y elaborado. Actualmente cuenta con una gran difusión, sobre todo en las esferas de la enseñanza, debido a que en un principio nació como un lenguaje con fines educativos. Por esta misma razón presenta ciertas limitaciones que lo alejan de ser un lenguaje idóneo para ciertas categorías de aplicaciones muy específicas. La filosofía del PASCAL se resume en el objetivo de conseguir que la programación se convierta en una tarea muy sencilla y sistemática, de forma que la creación de programas se pueda realizar poniendo en práctica métodos muy sencillos de comprender y recordar. Es por ello que el repertorio de instrucciones del PASCAL no es tan exhaustivo como puede ser el de otros lenguajes, como el BASIC o el LOGO; no obstante, estos últimos oponen a su naturaleza de lenguajes aptos para un mayor número de aplicaciones, una superior complejidad en la codificación de los programas. A pesar de todo, el PASCAL cuenta en nuestros días con una gran aceptación, incluso en el marco de la creación de software profesional.



*Los punteros señalan a una posición de memoria que contiene un dato de interés para el programa.*

| TABLA DE COMANDOS-PASCAL (1) | | |
|---|---|---|
| **Comando** | **Cometido** | **Observaciones** |
| TYPE <id.>=RECORD<lista> END; | Definición del tipo registro | Zona de declaraciones y definiciones |
| <id.1>:<tipo>; ...; <id.n>:<tipo>; | Definición de los campos de un registro (lista de campos) | Zona de declaraciones y definiciones |
| <id. registro>.<id. campo>; | Selector de campos del registro | Cuerpo del programa |
| WITH <var.> DO <sent.>; | Manejo de campos sin utilizar el identificador del registro | Cuerpo del programa |
| TYPE <id.>=FILE OF <tipo>; | Definición del tipo fichero | Zona de declaraciones y definiciones |
| RESET (<fichero>); | Inicializar fichero para lectura | Cuerpo del programa |
| EOF (<FICHERO>); | Indicador del fin de fichero | Cuerpo del programa |
| REWRITE (<fichero>); | Inicializar fichero para escritura | Cuerpo del programa |
| EOLN (<fichero>); | Indicador de fin de línea | Cuerpo del programa |
| <fichero>↑; | Variable «buffer» de un fichero | Cuerpo del programa |
| PUT (<fichero>); | Traslado al fichero de la variable «buffer» | Cuerpo del programa |
| GET (<fichero>); | Traslado a la variable «buffer» de la componente actual del fichero | Cuerpo del programa |
| TYPE <id.>=PACKED <tipo>; | Empaquetamiento de los datos para optimización de la memoria | Definición de los tipos: RECORD o ARRAY |
| TYPE <id.>=RECORD...CASE<dis.>OF<lista1>; | Definición de registro variante | Zona de declaraciones y definiciones |
| TYPE <id.>=^<tipo>; | Definición del tipo puntero | Zona de declaraciones y definiciones |
| NEW (<punt.>); | Inicialización de un puntero | Procedimiento estándar |
| DISPOSE (<punt.>); | Eliminación de un puntero | Procedimiento estándar |

NOTA: <id.>: identificador válido PASCAL. <lista>: lista de campos. <tipo>: cualquier tipo de datos, incluso estructurado. <var.>: variable tipo registro. <sent.>: sentencia simple o compuesta. <fichero>: identificador de una variable tipo fichero. <dist.>: campo distintivo como selector del CASE. <lista1>: lista de los valores del CASE que indiquen los campos en cada caso. <punt.>: variable de tipo puntero.

La principal característica, y a la vez el mayor éxito del PASCAL, reside en su naturaleza de lenguaje estructurado, tanto por lo que respecta al programa como a los datos. Esto hace que la legibilidad y flexibilidad de modificación de los programas en PASCAL sea, con mucha diferencia, superior a la de otros lenguajes no estructurados. También contribuye a esta sencillez y claridad el formato libre de escritura de los programas. Así, con un simple vistazo será posible comprender fácilmente la misión encomendada al programa, y conocer qué funciones realizan cada una de sus zonas; no hay que olvidar que otra de las características importantes del PASCAL es su modularidad, esto es: un programa en PASCAL se puede dividir en pequeños módulos o segmentos más fáciles de manejar y comprender.

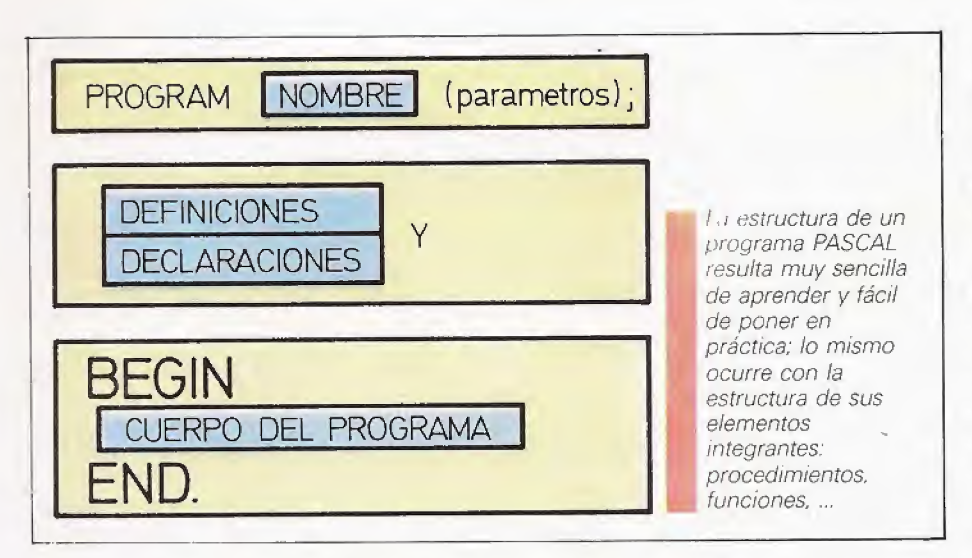
También se puede considerar como una peculiaridad el hecho de que un programa fuente escrito en PASCAL siempre debe ser traducido por medio de un compilador, y no a través de un intérprete. La compilación creará el denominado programa objeto, escrito en lenguaje máquina, que podrá ser ejecutado directamente siempre que no evidencie ningún tipo de error.

El hecho de tener que someterse a un proceso de compilación hace que resulte obligada la declaración de variables y demás elementos que vayan a ser utilizados posteriormente en el programa; de esta forma, el compilador reservará el espacio de memoria necesario para dichas variables, lo que garantizará el correcto funcionamiento del programa.

Las últimas tendencias en las técnicas de programación se ven reflejadas en el PASCAL; un lenguaje relativamente moderno. Así, por ejemplo, se presta una gran atención a la estructura de los datos y a su definición abstracta; como se señaló al principio, ello contribuye a la legibilidad del programa y facilita la explotación de la componente dinámica de éstos. La manipulación de los datos se ve favorecida, además, por su diferenciación en «tipos», lo que permite la detección de errores difíciles de descubrir de otro modo.

Los tipos de datos, clasificables en estructurados y no estructurados, se definen mediante la declaración de tipo TYPE. En PASCAL, es posible utilizar cuatro tipos de datos escalares estándar: REAL, INTEGER, BOOLEAN y CHAR, todos ellos no estructurados. Los tres primeros tipos permiten la manipulación de datos numéricos y lógicos de la forma convencional, mientras que el tipo CHAR está destinado al manejo de caracteres alfabéticos, numéricos y signos especiales.

Además de estos tipos, siempre disponibles en cualquier compilador, el PASCAL permite al usuario la definición de sus propios tipos de datos, según dos procedimientos: por enumeración de los posibles valores que podrá adoptar una variable de tal tipo, y como subrango o




*La estructura de un programa PASCAL resulta muy sencilla de aprender y fácil de poner en práctica; lo mismo ocurre con la estructura de sus elementos integrantes: procedimientos, funciones, ...*

subconjunto de los valores de otro tipo previamente definido.

La posibilidad de estructuración de los datos es la que otorga una nueva dimensión a este lenguaje. Así, cuenta con los siguientes tipos estructurados: ARRAY, SET, RECORD y FILE, además de con la posibilidad de crear estructuras dinámicas mediante el empleo de las variables de tipo puntero.

La declaración de las variables se efectúa siempre después de la definición de los tipos, y mediante la palabra reservada VAR. Las constantes tienen también un tratamiento especial en PASCAL: se pueden declarar mediante la palabra CONST lo que permitirá hacer referencia a una constante mediante un identificador. Esto último aporta la ventaja de que si dicho valor constante se quiere modificar en un momento dado, sólo habrá que modificar el valor asignado a su identificador, sin tener que tocar el resto del programa; de nuevo sale a relucir el carácter sistemático y clarificador del PASCAL.

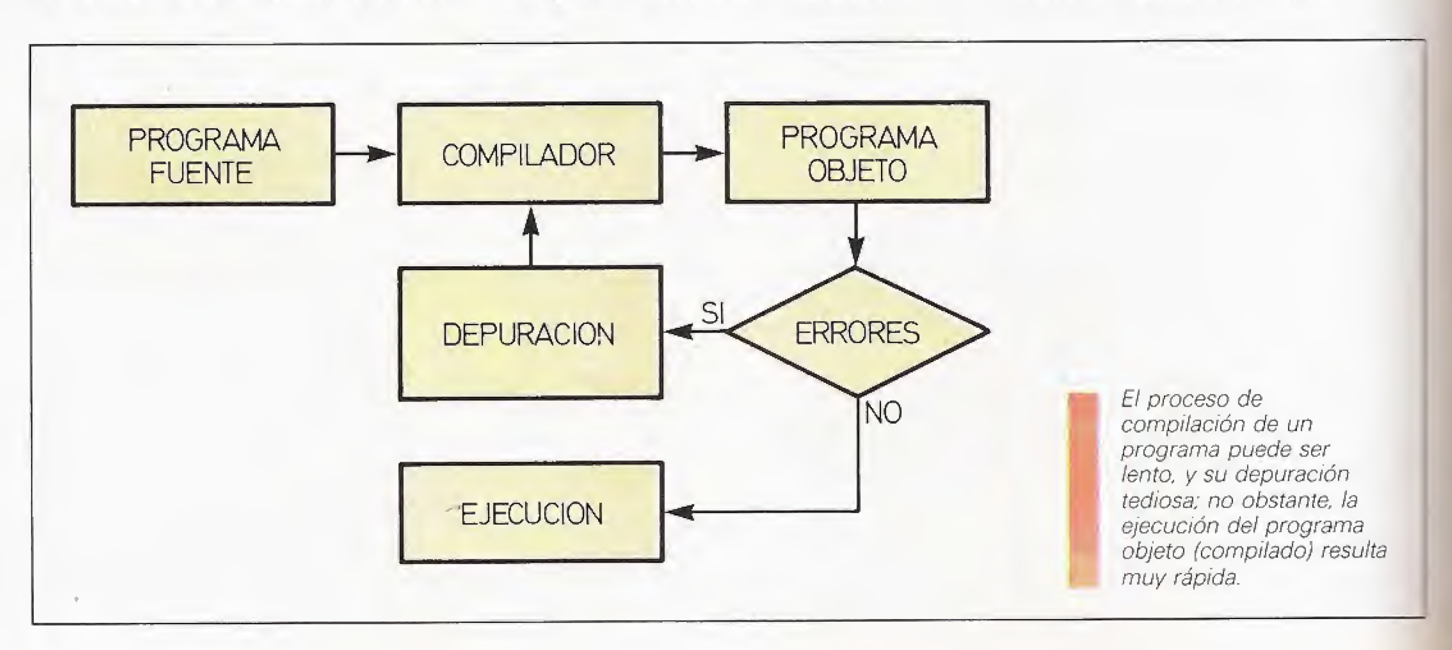
Todos los elementos del PASCAL mantienen la misma estructura: una cabecera, una zona de declaraciones, y, finalmente, el cuerpo principal de sentencias.

Incluso un programa completo mantiene esta misma estructura, de tal forma que cabe considerarlo como un superprocedimiento. A su vez, cada uno de los procedimientos es, realmente, un miniprograma con entidad propia. Las funciones conservan también esta estructura.

En definitiva, la cabecera determina la naturaleza de cada elemento, además de otorgarle un identificador y detallar los parámetros necesarios para la comunicación del mismo con el resto del programa, e incluso con el exterior. La zona de declaraciones se encarga de compendiar todos los elementos que se utilizarán posteriormente en el cuerpo principal del programa; esto irá encerrado entre las palabras BEGIN y END. Las referidas palabras se utilizan también para englobar sentencias simples en una única sentencia denominada compuesta; de ahí que también pueda considerarse al propio cuerpo de programa como una sentencia única.

Todo el «peso» del programa recae en la zona de declaraciones, en la que se definen los procedimientos y funciones; éstos son los principales elementos para la ejecución de acciones, hasta el punto que la mayor parte de las instrucciones del PASCAL son procedimientos o funciones estándar (READ, RESET, NEW, WRITE, INT, SIN...).

Realmente, la programación en PASCAL no resulta tediosa ni aburrida, y mucho menos a partir del momento en el que se ha llegado a una cierta familiaridad con la práctica. Su empleo crea además hábitos de programación muy recomendables, eficaces y aplicables a otros ámbitos de la informática.




*El proceso de compilación de un programa puede ser lento, y su depuración tediosa; no obstante, la ejecución del programa objeto (compilado) resulta muy rápida.*

# Prolog

## El lenguaje de la Quinta Generación



El nacimiento del Prolog supuso la aparición de una forma de programar que contrastaba sobremanera con lo que hasta el momento había sido la norma habitual en los lenguajes de programación. La programación lógica (el nombre «Prolog» está formado por las iniciales de los vocablos franceses «PROgrammation LOGique») es la nueva herramienta de que dispone la ingeniería del software para resolver cierto tipo de problemas que, hasta el momento de la llegada del Prolog, eran muy difíciles de representar. La historia de este lenguaje es muy corta; sus orígenes se remontan al año 1972 en la universidad de Marsella, en el laboratorio de programación de Alain Colmerauer.

Prolog es el máximo exponente de lo que es un «lenguaje declarativo». Sin excepción, con anterioridad a Prolog, todos los lenguajes de programación se encuadraban dentro de los denominados «lenguajes imperativos». Un programa escrito en un lenguaje de este último tipo (como puede ser el Basic) contiene una serie de comandos que especifican las acciones que hay que tomar en cada punto de la ejecución, en un orden rígido y claramente prefijado en función de los datos de entrada. La lectura de uno de estos programas nos proporciona una información precisa sobre la «conducta» del mismo.

Aunque muchas veces interese conocer a fondo la forma en la que un programa funciona —conocer su conducta—, lo normal ante un listado de 1.000 líneas en cualquier lenguaje es preguntarle a su autor: ¿Qué hace este programa?, en lugar de: ¿Cómo realiza su tarea este programa? A esta segunda pregunta contestaría de maravilla el simple listado del programa escrito en un lenguaje imperativo como los que conocemos.

La pregunta «¿qué hace este programa?» se debería contestar idealmente en términos de los datos de salida producidos por una determinada entrada. Lo que interesaría en este caso sería conocer la relación que liga a la entrada con los resultados, así como las relaciones que sirven de intermediarias entre la relación básica.




*Un programa escrito en Prolog consta tanto de relaciones como de reglas, y tanto unas como otras se albergan en una base de datos que mantiene el intérprete.*

La novedad que introdujo Prolog, es que se trata de un lenguje declarativo en el sentido de que los programas escritos en este lenguaje son fundamentalmente definiciones descriptivas de una serie de relaciones y funciones que hay que manejar. La diferencia entre un lenguaje imperativo y uno declarativo consiste, para expresarlo en pocas palabras y de una forma simplista, en que con los primeros estamos expresando «cómo» hay que resolver el problema, mientras que en un programa de los segundos se establecen las relaciones que expresan «qué» hay que hacer.

## El problema de la sintaxis

El interés que desde el principio suscitó el lenguaje Prolog entre los investigadores de Inteligencia Artificial ha sido la causa de la aparición de tres sintaxis ampliamente difundiadas del mismo: la original de Marsella, la de la Universidad de Edimburgo en Escocia y la que se encuentra en la implementación del lenguaje conocida como «micro-Prolog», del Imperial College de Londres.

Este lenguaje ha sido desarrollado fundamentalmente en Europa, en una reacción del viejo continente por tomar una parte activa en los problemas de la Inteligencia Artificial que, hasta el momento de su aparición, estaban dominados por la omnipresencia del Lisp, de origen norteamericano. El impulso definitivo puede venir de la mano del proyecto japonés de la Quinta Generación de Ordenadores, el cual ha tomado al Prolog como herramienta básica de trabajo. Todavía es algo pronto para decidir el éxito de este programa, del cual, como hemos dicho, depende en gran medida el prestigio que el lenguaje de Marsella sea capaz de adquirir a nivel internacional. De cualquier forma, e independientemente de los resultados prácticos, Prolog ha sido capaz de afian-

IGUAL

VECES 10

5 MENOS

VECES 4

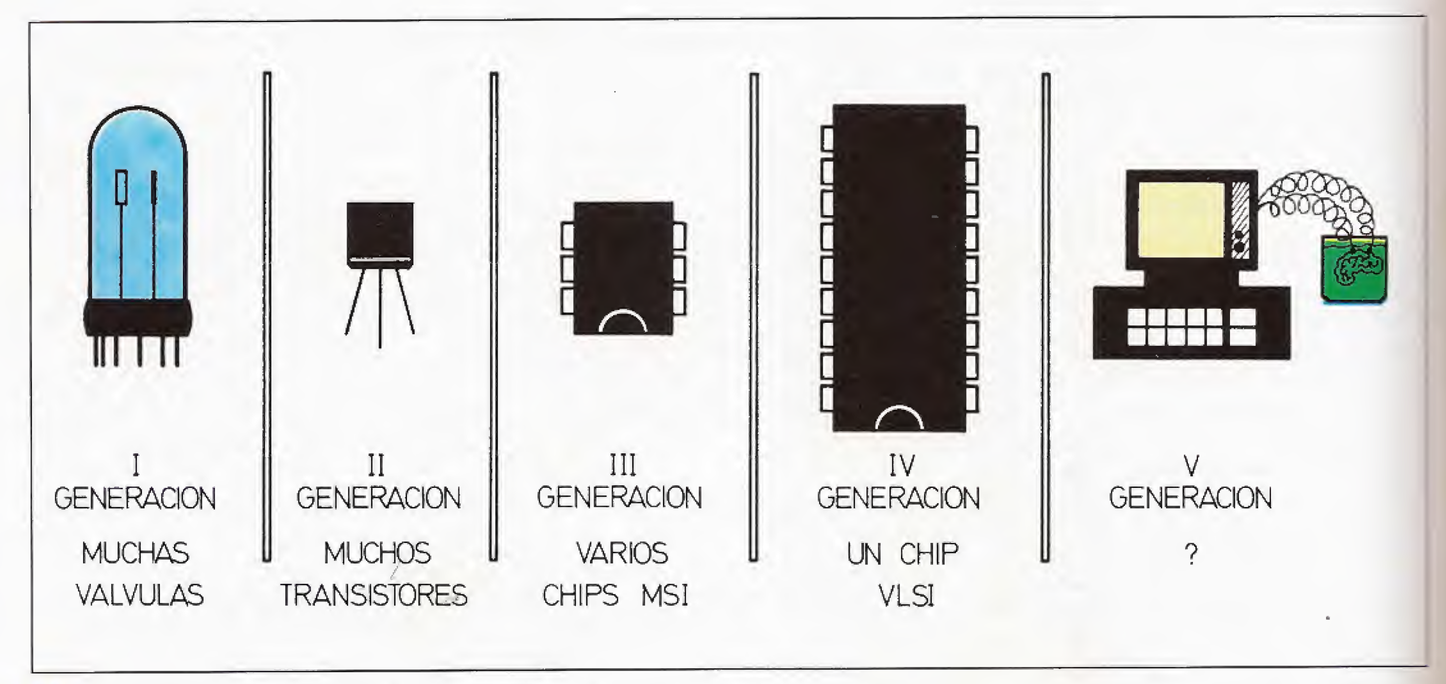
2 3

10 = 5 x ((2 x 3) - 4)

COMIDA

ENSALADA FILETE FRUTA

*El Prolog almacena los conocimientos relacionales en forma de árboles etiquetados, en los que cada nudo tiene una etiqueta. En la figura se observa cómo es posible representar una igualdad matemática y cómo expresar el hecho de que una comida está compuesta por tres platos.*

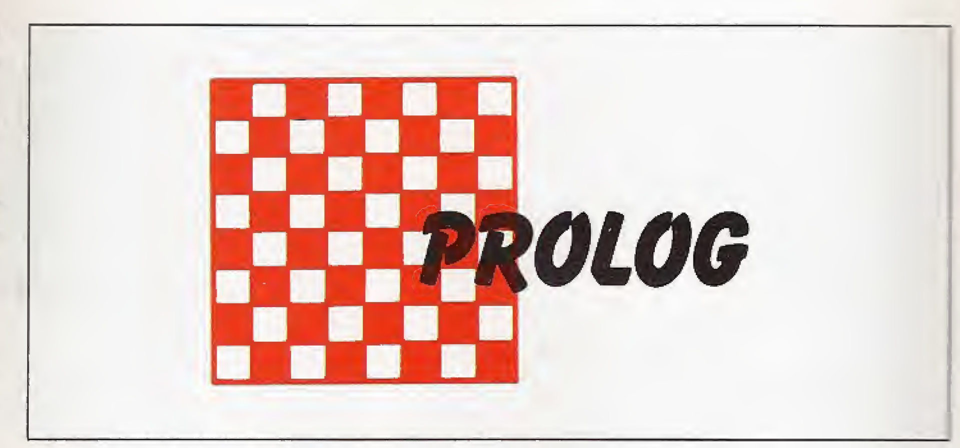



*De la válvula a la máquina inteligente en cinco generaciones de ordenadores.*

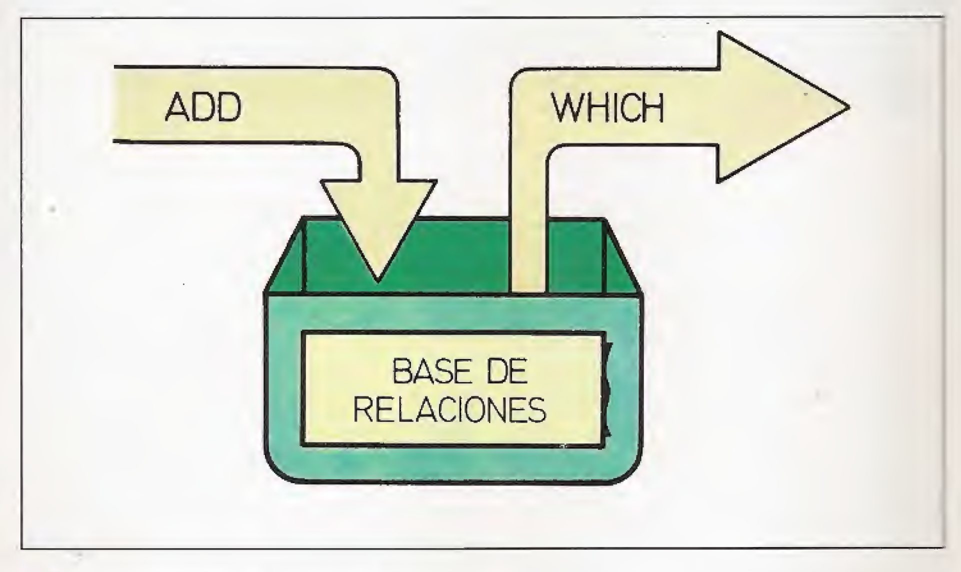
zar definitivamente los principios de la programación lógica en los poco más de diez años de su existencia.

La reciente popularización del Prolog ha estado condicionada en gran medida por la aparición de una serie de compiladores e intérpretes para el lenguaje capaces de operar en un entorno de PC, o incluso en máquinas tan modestas como el Spectrum o el BBC —un ordenador doméstico de gran difusión en Inglaterra—.

La aparición del micro-Prolog responde al intento de crear un vehículo de enseñanza de la programación y de introducción a la informática general, de forma que el ordenador haga acto de presencia en las aulas escolares. Su sintaxis es sin duda la más sencilla y fácil de aprender de las tres existentes, por lo que será la que aquí adoptemos para presentar las bases del lenguaje. Ha de quedar bien claro que será difícil reconocer un programa escrito con la sintaxis de Edimburgo si tan sólo se conoce la del micro-Prolog, pero los principios de la programación lógica aprendidos con ésta última son perfectamente aplicables a la primera sin más que cambiar los esquemas de escritura.




*El ajedrez es uno de los juegos en los que Prolog tendría mucho que decir.*




*Los componentes fundamentales del Prolog son la base de relacionales y los comandos básicos para acceder a ella.*

## Primeros pasos

Una implementación de Prolog con la sintaxis de micro-Prolog es, con casi total seguridad, interpretada. En realidad, lo que se tiene es un intérprete general de Prolog sobre el que se «superpone» un programa —también escrito en Prolog— que traduce las expresiones tecleadas según la sintaxis de micro-Prolog en otras más acordes con lo que se puede denominar un «prolog interno» en el que la ejecución es más eficiente. Ambos «Prologs» son independientes el uno del otro, por lo que el programador experimentado es capaz de escribir, pongamos por caso, el software necesario para convertir a su intérprete de Prolog en otro que trabaje con la sintaxis de Edimburgo.

En Prolog (nos referiremos a menudo a «Prolog» queriendo decir «micro-Prolog», es especial en las ocasiones en las que se comenten principios generales del lenguaje y no particularidades de cada sintaxis) el elemento fundamental es la existencia de una pequeña base de datos en la que se van almacenando los hechos y reglas a medida que van siendo introducidos en el sistema.

El ejemplo más característico del manejo de la base de datos en Prolog es el de las relaciones familiares, por lo que no vamos a ser menos en esta obra y empezaremos por describir este tipo de relaciones también. En una familia hay un buen número de hechos que relacionan los individuos aislados. Por ejemplo:

Enrique es el padre de Enriquito
Enrique es el padre de María

expresan dos relaciones de paternalidad, las que hay entre Enrique y sus hijos Enriquito y María. Estas dos frases son prácticamente un pedazo de programa en Prolog. En este lenguaje, una forma de representar hechos es a través de relaciones con tres componentes, cuya estructura es la siguiente:

<nombre-de-individuo>
<nombre-de-relación>
<nombre-de-individuo>

por lo que las dos relaciones vistas se podrían escribir de la siguiente forma:

Enrique padre-de Enriquito
Enrique padre-de María



*La Inteligencia Artificial es uno de los aspectos de la informática que más utilización hace del Prolog, como medio de expresión y formalización del conocimiento que parecen poseer los programas que «presentan» un comportamiento inteligente.*

Para añadir estas relacionea a la base de datos basta encerrarlas entre paréntesis y añadir la palabra «add», de la siguiente forma:

&.add(Enrique padre-de Enriquito)
&.add(Enrique padre-de María)

Los símbolos «&.» son la indicación que hace el intérprete para señalar que está esperando una entrada (es algo parecido al «Ready» del Basic). En Prolog los espacios en blanco son separadores, de ahí la necesidad de incluir un «-» en el nombre de la relación arriba expresada, ya que de otra forma estaríamos introduciendo al Prolog una relación que involucra a cuatro términos.

Las relaciones que hemos visto se caracterizan porque el nombre de la relación («padre-de») se encuentra entre los términos relacionados. Esta notación se llama por tal motivo «infija». También es posible encontrar relaciones que liguen no sólo a dos elementos, sino a tres o más. En tal caso la relación se ha de representar en «forma prefija», como es el caso de:

dar(Enrique María el-libro-rojo)

que expresa la relación que existe entre Enrique, María y el libro rojo. El apelativo de prefija es debido a que el nombre de la relación precede a los de los relacionandos. La adición de esta relación a la base de datos se haría encerrándola a toda ella entre paréntesis, a través del correspondiente «add».

Por último, hay relaciones que involucaran a un solo término. Esto es lo que ocurre cuando afirmamos que Enrique es varón, lo cual se representa en Prolog así (en forma «postfija» por ir el nombre de la relación en último lugar):

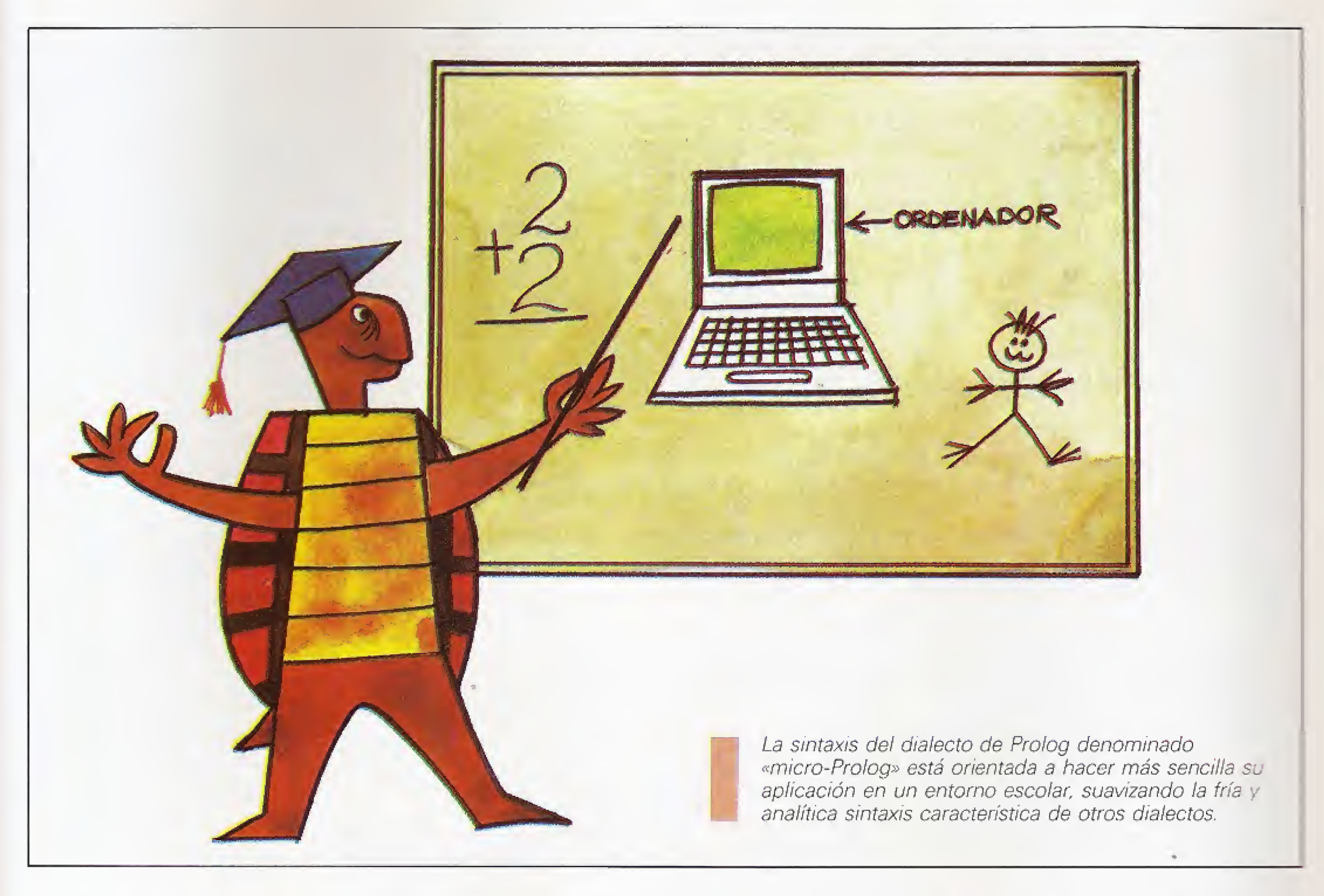
Enrique varón

Cualquier relación se puede reducir a la forma prefija en Prolog, lo cual es lo más habitual. Así, los ejemplos anteriores se escribirían:

padre-de(Enrique Enriquito)
padre-de(Enrique María)
dar(Enrique María el-libro-rojo)
varón(Enrique)




*Una de las razones para la rápida popularización del lenguaje Prolog ha sido la facilidad con la que es posible construir un intérprete para este lenguaje que puede aprovechar al máximo la escasa memoria que habitualmente se encuentra en los ordenadores personales.*




*La sintaxis del dialecto de Prolog denominado «micro-Prolog» está orientada a hacer más sencilla su aplicación en un entorno escolar, suavizando la fría y analítica sintaxis característica de otros dialectos.*

## Consultando la base de datos

Una vez visto la forma en la que se introducen relaciones en la base de datos, es el momento de recuperar la información almacenada.

Supongamos que se han introducido las siguientes relaciones:

```
&.add(Enrique padre-de Eduardo)
&.add(Enrique padre-de Isabel)
&.add(Isabel madre-de Enriquito)
&.add(Ana madre-de Isabel)
```

En este momento tenemos en la base de datos (también llamada a veces «base de relaciones» por razones obvias) cuatro relaciones entre diversos sujetos. Para comprobar la existencia de cualquiera de ellas se utiliza un predicado basado en «is», como a continuación se expresa:

```
&.is(Enrique padre-de Eduardo)
YES
&.
```

La contestación a la pregunta (YES) aparece en la línea siguiente a la pregunta. Si se hubiera dicho «is(Enrique abuelo-de Enriquito)» la contestación hubiera sido «NO», por no aparecer de forma explícita la relación a la que hacemos referencia, pese a que a partir de estos datos dicha relación se encuentra implícita.

Otra forma de consulta, basada en el «which», permite obtener valores concretos de los relacionandos. Considerando que el contenido de la base de datos es el anteriormente señalado, a continuación se expresa un acceso típico junto con su resultado:

```
&.which(x : Enrique padre-de x)
Eduardo
Isabel
No (more) answers
&.
```

El argumento del «which» está compuesto por dos partes. En la primera, la que aparece con anterioridad a los dos puntos (:), aparecen una serie de variables (una en nuestro caso) cuyo contenido ha de ser impreso a medida que se vaya recorriendo la base de datos en busca de predicados que coincidan en forma con la pregunta, la segunda parte de la cláusula «which», que en el caso precedente es «Enrique padre-de x». A la vista de un «which», el intérprete recorrerá la base de datos buscando relaciones que sean como la dada, siendo las variables que aparezcan una especie de «comodines» que van tomando valores a medida que progresa la búsqueda, imprimiéndose a continuación estos valores. Una vez que se ha recorrido toda la base de datos, aprece el mensaje «No (more) answers» para indicar la no existencia de nuevos resultados. Este mensaje aparecerá siempre que se finalice una exploración de la base de datos y, en algunos casos puede ocurrir que




*Si bien es posible encontrar versiones de Prolog corriendo sobre equipos domésticos, sólo con la mediación de un ordenador personal que soporte el sistema operativo MS-DOS es posible lanzarse a la creación de aplicaciones serias con este lenguaje.*

no se hayan encontrado ningún valor para las variables involucradas, de ahí el que aparezca entre paréntesis el «more».

En micro-Prolog, las variables son cualquier conjunto de caracteres que comiencen por las letras x, y o z. Cuando hay más de tres variables en un acceso a la base de datos, la costumbre es añadir un número al final de cada nombre de variable, de forma que todas queden inequivocamente definidas.

La parte que aparece a la izquierda de los dos puntos se denomina «patrón de respuesta». El significado de este nombre se ve más claro con el siguiente ejemplo:

```
&.which(x "es la madre de" y : x madre-de y)
Isabel es la madre de Enriquito
Ana es la madre de Isabel
No (more) answers
&.
```

Como en el caso anterior, la parte que se encuentra a la derecha de los dos puntos determina los valores que van tomando las variables «x» e «y» a medida que se recorre la base de datos. La parte izquierda de los mismos determina la forma en la que se han de presentar los resultados de la búsqueda: primero una variable, después un string de caracteres y, a continuación, el valor de la segunda variable. Es posible insertar en este string distintos códigos de control que tengan efectos diversos sobre el dispositivo impresor.

## Las reglas, principal característica del Prolog

Hemos visto cómo es posible introducir una serie de relaciones en la base de datos y obtener respuestas sobre la verdad de ciertas preguntas. Hasta el momento, todas las citadas preguntas se referían a relaciones que estaban presentes explícitamente en la base de hechos, y cualquier pregunta que no fuera contrastable directamente con una de tales relaciones daría resultados negativos. Sin embargo, ya hemos comentado que, al introducir relaciones sencillas entre elementos, es posible que aparezca alguna relación implícita entre ellos, la cual, desgraciadamente, siem-

pre sería contestada negativamente como acabamos de decir.

Así ocurre entre las cuatro relaciones sobre las que han versado los ejemplos anteriores. Al establecer que «Ana madre-de Isabel» y que «Isabel madre-de Enriquito», se está declarando implícitamente que Ana es la abuela de Enriquito, pero al preguntar "is(Ana abuela-de Enriquito)", la respuesta sería un rotundo NO. Por otra parte, pensemos lo que supondría la necesidad de tener que expresar todas y cada una de las posibles relaciones entre los elementos de una base de datos. Por pequeño que fuera su tamaño o escaso el número de relaciones, el trabajo pronto se escapa de la capacidad humana. A nuestra ayuda acuden en tales casos las reglas.

Las reglas se añaden a la base de datos de igual forma que las relaciones, a través de «add»:

```
&.add(x abuela-de y if
      x madre-de z and
      z madre-de y)
```

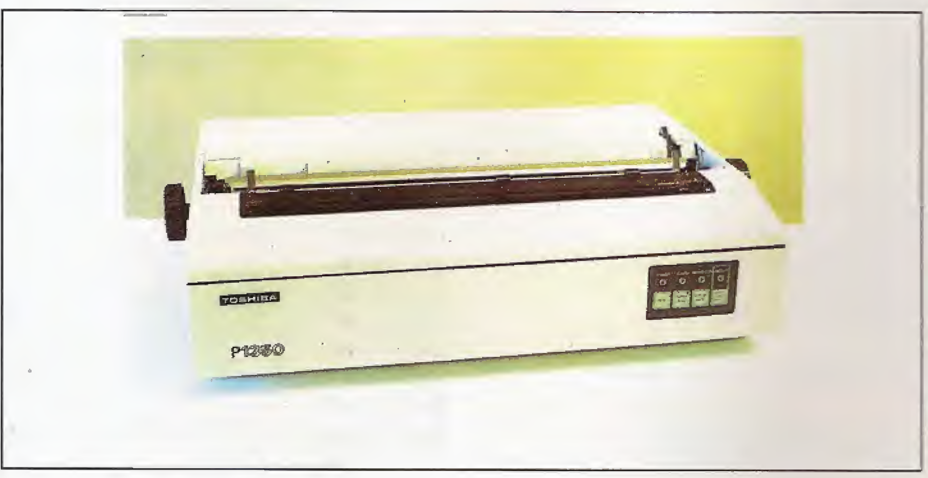
En términos más usuales, la regla anterior expresa el hecho de que si existe un «x» que es madre de un «z», y si este «z» es a su vez madre de un «y», entonces «x» resulta ser la abuela del «y».

Una vez que esta regla ha sido introducida en la base de datos, se obtendría el resultado esperado a la siguiente pregunta:

```
&.is(Ana abuela-de Enriquito)
YES
```

La forma en la que se expresan las re-

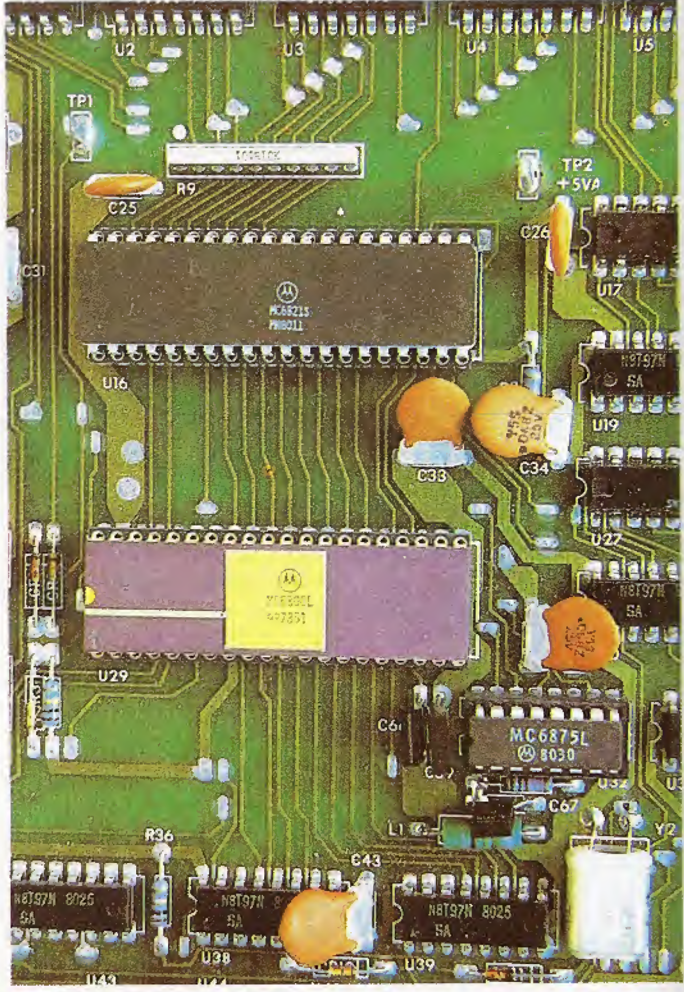
*La robótica es otro de los campos de la Inteligencia Artificial en el que el Prolog tiene mucho que aportar.*




*La comunicación con dispositivos periféricos en la mayoría de las implementaciones de Prolog se realiza a través de «streams» o canales de comunicación, hacia o desde los que se recoge toda la salida o entrada respectivamente al ser seleccionados por defecto.*




*La posibilidad de conectar dispositivos periféricos avanzados al ordenador y controlarlos desde un programa escrito en Prolog, es una facultad que sólo depende del hardware del equipo que se esté manejando.*




*Como ocurría con las implementaciones del lenguaje Lisp, las de Prolog son igual de ineficientes al ser ejecutadas sobre arquitecturas convencionales de ordenadores.*

glas es uno de los puntos que más distancian las distintas sintaxis del lenguaje Prolog. Concretamente, según la sintaxis de Edimburgo, la misma regla tendría el siguiente aspecto:

```
abuela-de(X,Y):-
        madre-de(X,Z),
        madre-de(Z,Y).
```

El cual puede provocar no pocos sustos si no se conoce la particular forma de representación. Lo importante es que los mismos principios son aplicables en cualquiera de los casos.

Todo programa en Prolog no es sino una colección de hechos y de reglas en una base de datos común, la cual es consultada por el usuario. Por lo tanto, con los elementos vistos hasta el momento se estaría en disposición de comenzar cualquier proyecto con este lenguaje. Sin embargo, otra de las grandes habilidades de Prolog reside en la facilidad con la que se manejan listas, lo que abre las puertas a la manipulación de complicadas estructuras de datos.

## Las listas en Prolog

Una lista no es más que un conjunto de palabras encerradas entre paréntesis y separadas por espacios en blanco. Por ejemplo, la lista (fútbol tenis baloncesto) puede representar los deportes practicados por alguna persona. Con estos elementos, se puede decir:

```
&.add(juan practica (fútbol tenis baloncesto))
```

Las relaciones familiares proporcionan de nuevo buenos ejemplos para ver la utilidad de las listas. La siguiente estructura general:

```
(el-padre la-madre) son-padres-de (sus hijos)
```

da pie para la representación de los siguientes núcleos familiares:

```
(Enrique Berta) son-padres-de (Margarita Carlos)

(Juan María) son-padres-de (Isabel Pablo Luis)

(Pepe Marta) son-padres-de ()

(Jose-Luis Olga) son-padres-de (Teófilo)
```

De ellas se desprende que Pepe y Marta no tienen hijos, mientras que José Luis y Olga tan sólo tienen uno.

Una vez introducidas las anteriores relaciones a través de «add» en nuestra base de datos, es posible contestar a la siguiente pregunta:

```
&.which(x : (Juan María) son-padres-de x)
(Isabel Pablo Luis)
No (more) answers
&.
```

Observemos que la respuesta viene dada entre paréntesis, indicando que se trata de una lista.

Las listas son un medio conveniente para representar la información, como hemos podido comprobar en los ejemplos anteriores. A base de listas constituidas a su vez por listas es posible representar casi cualquier estructura de datos que se nos venga a la cabeza, sin más que conocer la forma en la que tal estructura está dispuesta sobre la lista.

Para un manejo eficiente y elegante de listas, el Prolog distingue entre la «cabeza» y la «cola» de las mismas. En una lista como «(fútbol tenis baloncesto)», «fútbol» es la cabeza, mientras que la cola está formada por la lista «(tenis baloncesto)». Es muy importante darse cuenta de que la cola es siempre una lista, mientras que la cabeza puede serlo o no. En un ejemplo como el anterior, la cabeza era un sólo elemento —«fútbol»—, pero en esta otra:

```
((a b) c d)
```

la cabeza sería «(a b)», mientras que la cola estaría formada por «(c d)».

La forma general de representar las cabezas y las colas en Prolog es a través de la barra vertical (|). Por ejemplo, al aplicar el patrón (x|y) sobre la lista anterior, resultaría en la variable «x» con el valor «(a b)» y la «y» con el valor «(c d)». Volviendo al ejemplo de las listas de padres e hijos, se pueden obtener el primer hijo del matrimonio entre Juan y María de la siguiente forma:

```
&.which(x : (Juan María) padres-de (x|y))
Isabel
no (more) answers
```

Este último ejemplo da pie para comentar el hecho de que a veces se necesitan variables con el simple propósito de albergar ciertos valores temporales, como es el caso de la «y» en el valor anterior. Su utilidad estriba en conseguir una pregunta sintácticamente correcta, pero su valor no es utilizado para nada. Algunos intérpretes permiten utilizar la variable «—» (underscore) como comodín para estos casos.

La combinación de la estructura lista junto con definiciones recursivas permiten resolver ciertos problemas de forma muy elegante. Tal es el caso que se encuentra al querer determinar si un elemento es miembro de una lista, un problema muy típico de Prolog y, en general, de cualquier lenguaje de programación orientado a listas. Para ello se establecerá una relación binaria entre un elemento y una lista y una regla, cuyo propósito es representar la siguiente condición de pertenencia de un elemento a una lista:

1.º Un elemento pertenece a la lista si el primer elemento de la misma es precisamente el elemento buscado.

2.º Un elemento pertenecerá a una lista si al no coincidir con el primer elemento de la misma, es miembro de su cola.

Las siguientes relaciones Prolog expresan las condiciones arriba expuestas:

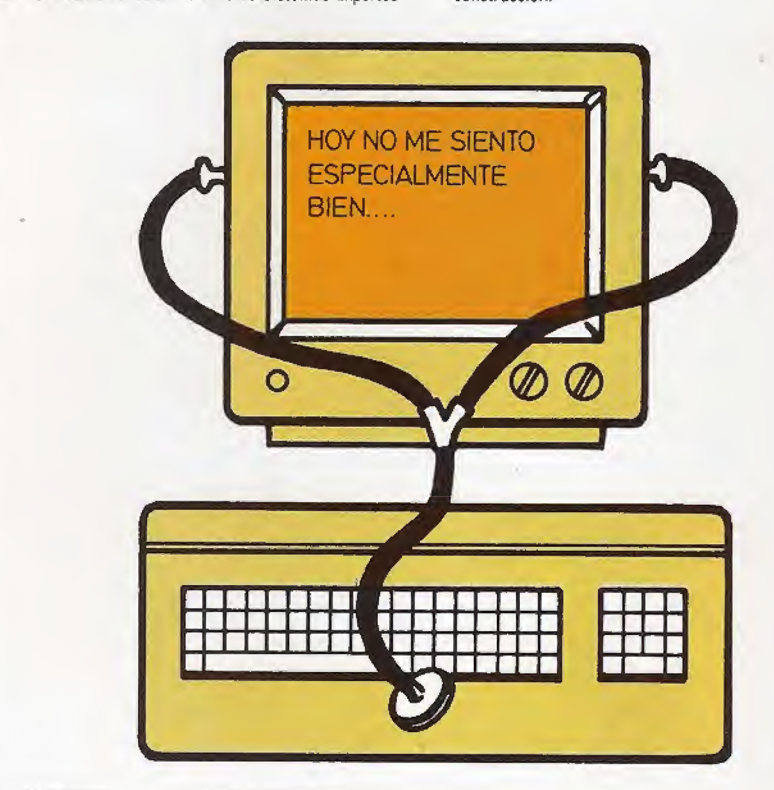
```
&.add(x pertenece-a (x|z))
&.add(x pertenece-a (y|z) if
        x pertenece-a z)
```

Al preguntar al intérprete de micro-Prolog:

```
is(1 pertenece-a (1 2 3))
```

## Los sistemas expertos

Otro aspecto al que Prolog se adapta especialmente bien es el campo de los Sistemas Expertos. Estos son, básicamente, programas capaces de almacenar y asimilar el conocimiento que previamente han aquirido de un experto humano en un área determinada, para que, con posterioridad, otras personas no tan especializadas puedan pedir consejo y recibir explicaciones del propio programa. Hasta la fecha se han construido un buen número de Sistemas Expertos que funcionan con mayor o menor fortuna en diversas áreas de conocimiento. Destacan por su cantidad y calidad los dedicados a aplicaciones médicas, como MYCIN, versado en la diagnosis de enfermedades infecciosas de la sangre, y ONCOCIN, experto en el tratamiento de algunos tipos de cáncer.
Los Sistemas Expertos se enfrentan con problemas de representación del conocimiento, de imprecisión en el mismo, de comprensión del lenguaje natural cuando hacen las preguntas los «no expertos», etc. El Prolog se perfila como la herramienta básica para su construcción.

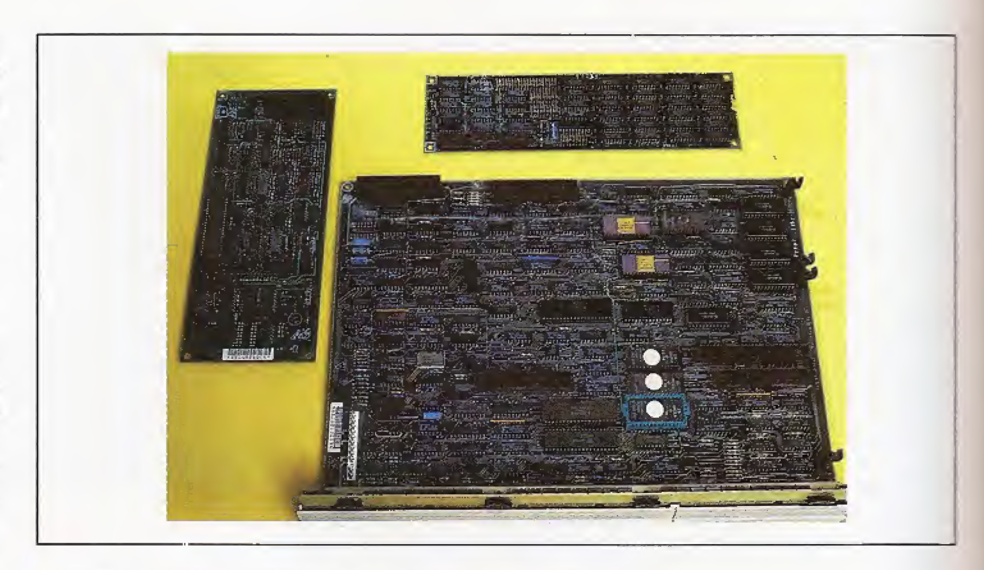


la respuesta será YES, ya que ha sido posible encontrar un hecho en la base de datos —«x pertenece-a (x|z)»— que es cierto al ser sustituidas las variables del mismo con los términos en los que se formula la pregunta. Por otra parte, al preguntar:

```
is(2 pertenece-a (1 2 3))
```

ya no se cumple el primero de los hechos, por lo que el Prolog intentará buscar una nueva relación que aplicar. En este caso, se encuentra con el enunciado de una regla (segunda de las relaciones). Concretamente, intentará demostrar «2 pertenece-a (2 3)», partiendo de nuevo de las dos relaciones de que dispone en la base de datos. En esta ocasión, sí es posible aplicar la primera de ellas, por lo que el resultado será de nuevo cierto (YES).

Este tipo de definiciones recursivas tienen propiedades muy curiosas, entre las cuales la más sobresaliente es la posibilidad de que funcionen de forma inversa a como se ha visto; es decir, en lugar de contestar positiva o negativamente a una pregunta, ser capaces de generar los elementos de los que se compone la lista forzando el hecho de que se desea un resultado cierto. Este tipo de utilización de estas definiciones caen fuera del ámbito de los propósitos de esta introducción al Prolog, por lo que bastará comentar dicha posibilidad.



*Para resolver el problema de la ineficiencia del Prolog al ser ejecutado sobre arquitecturas convencionales de ordenador, se han lanzado varios proyectos a nivel mundial encaminados a la construcción de máquinas cuyas características estén más de acuerdo con los principios establecidos en el lenguaje y en las que, por tanto, el funcionamiento de aplicaciones escritas en Prolog sea más rápido y eficaz.*




*El acceso a las características de color del ordenador no está ni mucho menos estandarizado en el lenguaje Prolog. Cada implementación concreta ha optado por diversos medios de acceso a las capacidades gráficas de los controladores gráficos habituales de los ordenadores.*

## A modo de epílogo

Con los párrafos anteriores se ha pretendido dar una visión muy esquemática de los principios del lenguaje Prolog, y el estado actual de su desarrollo. La base matemática que impregna la ciencia de los ordenadores tiene un reflejo muy persistente en el Prolog, ya que este lenguaje se puede considerar como una auténtica implementación concreta de un tipo determinado de lógica matemática.

Como ocurre con el lenguaje Lisp, cualquier implementación de Prolog sobre un ordenador convencional es extremadamente ineficiente por el extensivo uso que de la estructura lista se hace. A diferencia de él, la realización de ordenadores cuya arquitectura esté especialmente diseñada para soportar aplicaciones escritas en el lenguaje de Colmerauer son sólo proyectos, los cuales se están llevando a cabo fundamentalmente en el Japón.

# Indice Temático

# Modula-2



# PASCAL (1)



# PASCAL (2)



# PASCAL (3)

## PASCAL (y 4)



## Prolog



**Cuadros**